

A Controlled Evaluation of Range-Sensitive Dataflow Coverage for Indirect Testing of Input Constraints in Cyber-Physical Systems

Daria Rudenko[†], Oleksandr Kolchyn^{†*}

V.M. Glushkov Institute of Cybernetics NAS of Ukraine, Glushkov ave. 40, 03187 Kyiv, Ukraine

Abstract

Cyber-physical systems rely on sensor inputs whose values vary continuously, and whose validity depends on the context of use. Conventional structural coverage criteria do not distinguish between different value ranges of such input-originated data and may therefore miss faults that occur only under specific input conditions. This paper presents a controlled empirical evaluation of a context-aware range-sensitive coverage, criterion that extends structural coverage by exercising distinct value ranges at use points. By linking input definitions to their downstream uses, the approach provides an indirect assessment of input handling and constraint enforcement. We evaluate the approach on a benchmark of CPS-inspired programs with injected faults affecting input validation and propagation. Test suites generated by established tools and large language models are compared under both structural and range-sensitive criteria. The results indicate that range-sensitive dataflow coverage improves fault detection for input-dependent logic compared to traditional criteria, while introducing manageable overhead.

Keywords

software testing, coverage criteria, input validation, test cases generation, cyber-physical systems

1. Introduction

Cyber-physical systems (CPS) [1, 2] rely on sensor inputs whose values vary continuously and whose validity depends on the context in which they are used [3]. To ensure safe operation, software components impose constraints on admissible input values through validation checks, such as range limits or consistency conditions across multiple sensors. However, conventional coverage-driven automated test generation techniques [4–7] primarily aim to maximize structural coverage criteria rather than systematically exploring the input space. As a result, test suites achieving high structural coverage may still fail to exercise critical regions of input values relevant to system behavior.

In this work, we focus on white-box testing approaches, where test generation relies solely on the structure of the program under test. While specification-based testing techniques provide complementary guarantees by deriving tests from formal models or requirements, they are outside the scope of this study.

Input validation logic embedded in program code introduces constraints on admissible values of program variables derived from external inputs (hereafter referred to as input constraints). These constraints restrict which values may propagate through the program and influence its behavior. Importantly, the effect of such constraints is determined not only by the validation checks themselves, but also by how validated values are subsequently used. In many cases, different value ranges lead to different behaviors at the program points where these values are consumed. Therefore, effective testing requires exploring the range of values that can reach such use points after passing through validation logic.

Traditional structural coverage criteria, such as branch coverage, modified condition / decision coverage (MC/DC) [8], and dataflow criteria [9] (e.g., All-Uses coverage [10]), measure how

2nd Workshop "Intelligent information technologies" UkrProg-IIT'2026 co-located with 16th International Scientific and Practical Programming Conference UkrPROG'2026, May 5-6, 2026, Kyiv, Ukraine

* Corresponding author.

[†] These authors contributed equally.

✉ dasharudenko1902@gmail.com (D. B. Rudenko); kolchin_av@yahoo.com (O. V. Kolchyn)

ORCID 0009-0008-7499-5895 (D. B. Rudenko); 0000-0001-7809-536X (O. V. Kolchyn)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

thoroughly control-flow and dataflow elements are exercised during testing. However, they do not account for the diversity of value ranges reaching these elements [11]. Consequently, achieving high structural coverage does not guarantee that test cases adequately explore input-dependent behavior.

The key idea of our approach [11, 12] is to exercise input constraints indirectly by ensuring coverage of equivalence classes of input-originated values at their points of use, rather than validating inputs in isolation. This perspective motivates the use of range-sensitive dataflow coverage (RS-DFC), which extends traditional All-Uses criterion by distinguishing value ranges propagated from input definitions to their downstream uses. While the concept of RS-DFC has been previously introduced [13], its practical effectiveness for detecting faults in input-dependent behavior has not yet been systematically evaluated.

This paper addresses this gap by providing a controlled evaluation of RS-DFC in the context of cyber-physical systems. We conduct experiments on a benchmark suite of CPS-inspired programs derived from real-world code fragments responsible for sensor input handling and validation, capturing representative patterns across multiple application domains.

The contributions of this paper are as follows:

- We provide a controlled evaluation of RS-DFC for testing input constraints in cyber-physical systems.
- We introduce a benchmark suite of CPS-inspired programs derived from real-world sensor input handling code.
- We design mutation operators modeling realistic faults in input constraint handling and propagation.
- We experimentally compare test suites generated by CBMC, and large language models under structural and range-sensitive coverage criteria.

The remainder of this paper is organized as follows. Section 2 reviews background and related work. Section 3 introduces the RS-DFC criterion. Section 4 describes the benchmark and mutation operators. Section 5 presents the experimental setup and methodology. Section 6 reports and discusses the results. Section 7 outlines threats to validity, and Section 8 concludes the paper.

2. Input Validation Challenges in Cyber-Physical Systems

CPS combine computational components with physical processes, creating unique challenges for input validation that do not appear in traditional software systems. Input validation checks, implemented through constraints on sensor inputs, must handle noisy, continuous, heterogeneous, and time-critical data from sensors, actuators, and other external devices.

Heterogeneous Input Sources. CPS receive inputs from multiple modalities, such as sensor readings (temperature, pressure, position, velocity), human-machine interfaces, networked devices or other CPS components. These inputs can vary in format, scale, and reliability. A single input validation method designed for structured software inputs (e.g., type checks or simple range checks) is often insufficient for heterogeneous CPS inputs, which may be noisy, delayed, or partially missing.

Example 1.

```
// Validation function with tolerance for sensor noise
bool validateTemperature(float temp, float nominal) {
    const float TOLERANCE = 0.5; // +/- 0.5°C tolerance for sensor noise
    return temp >= (nominal - TOLERANCE) &&
           temp <= (nominal + TOLERANCE);
}
bool validatePressure(float pressure, float nominal) {
    const float TOLERANCE = 100; // +/- 100 Pa tolerance for sensor noise
    return pressure >= (nominal - TOLERANCE) &&
           pressure <= (nominal + TOLERANCE);
}
// sensor readings
```

```

float temperature = readSensor();
float pressure = readSensor();
float nominalTemp = 22.0; // expected room temperature
float nominalPressure = 101325; // standard atmospheric pressure

if (!validateTemperature(temperature, nominalTemp) ||
    !validatePressure(pressure, nominalPressure)) {
    return ERROR; // fail-safe: do not proceed with invalid inputs
}

// Downstream usage: adjust heating/cooling based on validated sensor values
adjustHeatingCooling(temperature, pressure);

```

This snippet illustrates heterogeneous CPS inputs with noise and variability. Temperature and pressure readings are validated against admissible ranges with a tolerance to account for sensor noise. Only validated inputs are propagated to downstream logic (`adjustHeatingCooling`), which controls the Heating, Ventilation, and Air Conditioning (HVAC) system. Indirect testing of input constraints requires generating test cases that exercise diverse value ranges including realistic sensor noise, ensuring the actuator behaves correctly under noisy CPS conditions with multiple sensor types.

Example 2.

```

// sensor readings
float temperature = readSensor(); // temperature reading
float timestamp = getSensorTimestamp(); // time when the reading was captured
float currentTime = getCurrentTime();
// Check for delayed data
if (currentTime - timestamp > MAX_DELAY) {
    return ERROR; // reading too old, do not use
}
// Validate sensor range
if (!validateTemperature(temperature)) {
    return ERROR; // out-of-range value
}
// Downstream usage: control heating/cooling system
adjustHeatingCooling(temperature);

```

This snippet illustrates CPS input handling with delayed sensor readings. The temperature value is checked for recency using timestamps; readings exceeding a maximum allowed delay are discarded to prevent unsafe actions. After passing the range validator, the reading is used to adjust a heating/cooling system. Indirect testing of input constraints requires exercising both the valid ranges of temperature and the possible timing delays, ensuring that downstream actuator logic behaves safely under realistic CPS conditions with delayed sensor updates.

Continuous and Real-Time Nature. Many CPS operate in real-time, processing continuous signals rather than discrete events. Input validation must therefore account for temporal properties (e.g., values arriving too fast or too slow), physical constraints (e.g., acceleration limits, control boundaries), dynamic correlations between multiple inputs. Traditional static or unit-test approaches typically assume discrete, deterministic inputs, which cannot capture violations in continuous-time signals or complex sensor correlations.

Example 3.

```

// Validator for continuous sensor input: acceleration limit
bool validateAcceleration(float prevSpeed, float currSpeed, float dt){
    float accel = (currSpeed - prevSpeed) / dt; // compute acceleration
    const float MAX_ACCEL = 5.0; // m/s^2, physical limit
    return fabs(accel) <= MAX_ACCEL;
}
// sensor readings
float prevSpeed = getPreviousSpeed(); // previous speed reading

```

```

float currSpeed = readSensor();           // current speed reading
float dt = getDeltaTime();                 // elapsed time since last reading
// Validate rate-of-change
if (!validateAcceleration(prevSpeed, currSpeed, dt)) {
    return ERROR; // acceleration exceeds physical limits
}
// Downstream usage: adjust vehicle throttle
updateThrottle(currSpeed, prevSpeed, dt);

```

This snippet illustrates handling of continuous, real-time CPS inputs. Vehicle speed is sampled at discrete time intervals, and the validator enforces a maximum acceleration limit, reflecting physical constraints of the vehicle. Only validated speed readings are propagated to the downstream actuator (`updateThrottle`). Indirect testing of input constraints requires exercising combinations of consecutive speed values and time intervals to cover possible accelerations, ensuring correct actuator behavior under realistic continuous operating conditions.

Safety-Critical and Robustness Requirements. CPS often control safety-critical systems (autonomous vehicles, industrial robots, medical devices). Input validation failures can lead to physical harm or property damage, not just software errors. Therefore it is not enough to verify logical correctness of code. Validation must account for physical plausibility, fault tolerance, and environmental uncertainty.

Example 4.

```

// Validator for a redundant temperature sensor
bool validateTemperatureBackup(float temp) {
    return temp >= -40.0 && temp <= 85.0;
}
// sensor readings: controlling a reactor heater
float tempPrimary = readSensor(); // primary temperature sensor
float tempBackup = readSensor(); // backup sensor
// Validate sensors
bool primaryValid = validateTemperature(tempPrimary);
bool backupValid = validateTemperatureBackup(tempBackup);
if (!primaryValid && !backupValid) {
    return ERROR; // fail-safe: both sensors invalid, shutdown heater
}
// Downstream usage: heater control based on validated readings
float tempToUse = primaryValid ? tempPrimary : tempBackup;
adjustHeater(tempToUse);

```

This snippet demonstrates safety-critical CPS input handling with redundant sensors. Two temperature sensors monitor a reactor heater. If both sensors are invalid, the system enters a fail-safe state and prevents actuator operation. If only one is valid, it is used for downstream control. Indirect testing of input constraints requires exercising scenarios where primary and backup sensors provide valid and invalid readings, ensuring robust downstream actuator behavior under fault conditions.

Complex Interactions and System Dynamics. Inputs in CPS rarely influence the system in isolation. Instead, non-linear interactions can amplify small input deviations, feedback loops can propagate errors, multi-component dependencies require system-level input validation. These dynamic and compositional aspects are generally ignored by “simple” software testing methods, which validate components independently.

Example 5.

```

// Validators for multiple sensors
bool validateLoad(float load) {
    return load >= 0.0 && load <= 100.0; // percent
}
bool validateAngle(float angle) {

```

```

    return angle >= -30.0 && angle <= 30.0; // degrees
}
// sensor readings: controlling a robotic arm
float loadSensor = readSensor(); // current load on arm
float angleSensor = readSensor(); // current arm angle
// Validate sensors individually
if (!validateLoad(loadSensor) || !validateAngle(angleSensor)) {
    return ERROR; // fail-safe: invalid input prevents movement
}
// Downstream usage: compute actuator force as a non-linear function of load and angle
float actuatorForce = (loadSensor/100.0)*(1.0 + pow(angleSensor/30.0, 2));
setArmForce(actuatorForce);

```

This snippet illustrates CPS input handling with complex interactions between multiple sensors. The robotic arm receives readings for load and angle, validated individually. The downstream actuator force is computed as a non-linear function of both inputs. Indirect testing of input constraints requires exercising combinations of load and angle values across their admissible ranges to ensure that downstream actuation responds correctly under realistic scenarios where multiple sensor values interact, potentially amplifying effects due to non-linear system dynamics.

Context-dependent nature of input constraints. The validity and interpretation of input values often depend on the operational mode or system state in which they are used. For example, admissible speed ranges may differ between steady-state operation, acceleration phases, and emergency conditions. Similarly, acceptable sensor readings may vary depending on environmental conditions or control objectives. As a result, input constraints cannot be adequately characterized by a single global range; instead, they must be understood as context-dependent partitions of the input space. Effective testing therefore requires not only covering the overall range of input values, but also exercising distinct value ranges under different usage contexts, where they may lead to different behaviors.

Example 6.

```

// Validator for vehicle speed depending on operating mode
typedef enum {NORMAL, ACCELERATION, EMERGENCY} Mode;
bool validateSpeed(float speed, Mode mode) {
    switch(mode) {
        case NORMAL:
            return speed >= 0.0 && speed <= 60.0; // km/h
        case ACCELERATION:
            return speed >= 0.0 && speed <= 120.0;
        case EMERGENCY:
            return speed >= 0.0 && speed <= 180.0;
        default:
            return false;
    }
}
// sensor readings: controlling vehicle throttle
float vehicleSpeed = readSensor(); // current speed reading
Mode currentMode = getCurrentMode();
// Validate speed based on current mode
if (!validateSpeed(vehicleSpeed, currentMode)) {
    return ERROR; // invalid speed for mode
}
// Downstream usage: adjust throttle according to validated speed
adjustThrottle(vehicleSpeed, currentMode);

```

This snippet illustrates context-dependent CPS input constraints. The vehicle speed sensor is validated according to the current operating mode: NORMAL, ACCELERATION, or EMERGENCY, each with different admissible ranges. Only validated speeds are propagated to downstream throttle control. Indirect testing of input constraints requires exercising value ranges for each operating

mode, ensuring that the system behaves safely and correctly under different dynamic contexts, reflecting real-world CPS scenarios where constraints depend on operational state.

Implications for Testing Approaches. First, coverage adequacy cannot be assessed solely by checking whether validation conditions are exercised; it also depends on whether relevant value ranges reach the points where they affect system behavior. Second, interactions between multiple inputs and constraints make it insufficient to consider values in isolation, as faults may arise only under specific combinations or ranges of values. Third, the continuous nature of inputs makes exhaustive exploration infeasible, requiring abstractions such as equivalence classes of values. Because of the above challenges, CPS input validation requires methods that are:

- Environment- and context-aware – can handle continuous signals, sensor noise, and temporal correlations.
- Robust to heterogeneity – works across multiple sensor types, interfaces, and data formats.
- Coverage-oriented – measures how well test inputs explore unsafe or unexpected system behaviors.
- Independent of formal specifications – CPS may lack complete formal specifications for all physical interactions; evaluation needs to work directly on code or models.

Taken together, these aspects highlight the need for testing approaches that capture how input-originated values propagate through the program and influence behavior at their points of use. This motivates the use of range-sensitive dataflow coverage, which distinguishes classes of values along def-use chains and enables systematic exploration of input-dependent behavior in CPS.

3. Background and Related Work

In practice, input validation logic is typically embedded directly in program code and enforces constraints on admissible values of sensor readings and derived variables.

Common validation patterns in cyber-physical systems (CPS) include:

- (i) range checks for individual sensor readings;
- (ii) consistency checks across multiple sensors measuring related (or redundant) quantities;
- (iii) checks of dynamic signal properties, including rates of change and boundary limits;
- (iv) temporal consistency checks and dependencies on previous values (signal history);
- (v) completeness and freshness checks of data, including detection of missing or stale values;
- (vi) context-dependent checks that take into account the current operating mode or system state.

As illustrated in Section 2, such constraints are often context-dependent and may involve interactions between multiple inputs or temporal properties of continuous signals.

Existing approaches to testing CPS software address input-related faults in several ways. Structural testing techniques aim to systematically exercise program logic, while specification-based approaches derive tests from formal models or digital twins [18]. In addition, methods such as fuzzing [16], fault injection [17], and simulation-based scenario testing explore input variability by generating diverse or adverse conditions.

However, in practice, formal specifications are often unavailable, incomplete, or too abstract to capture all relevant physical constraints and input interactions. As a result, testing is additionally performed in a specification-less, code-driven manner, relying on structural coverage criteria or heuristics. In such settings, input validation is typically assessed only indirectly—through its effect on program execution—rather than by explicitly analyzing how different classes of input values propagate to their points of use.

Furthermore, existing approaches tend to focus on control-flow coverage, robustness under faults, or system-level behavior, rather than on systematically exploring semantically distinct value ranges induced by validation logic. As a consequence, important input conditions—such as boundary regions, combinations of sensor values, or context-specific constraints—may remain insufficiently tested even when traditional coverage metrics are satisfied.

These limitations motivate the need for testing approaches that explicitly account for how input-originated values are constrained, propagated, and ultimately used within the program. In the following sections, we review structural coverage criteria (Section 3.1) and automated test generation techniques (Section 3.2), and discuss their limitations in the context of CPS input validation.

3.1. Structural Coverage Criteria

Structural (white-box) coverage criteria are a family of metrics designed to evaluate how thoroughly a test suite exercises the internal elements of a program, focusing on control-flow and data-flow aspects. Unlike specification-based testing, structural coverage does not require formal requirements; instead, it relies on analyzing the code itself. Commonly used coverage criteria include:

- *Statement coverage*: measures the proportion of program statements executed at least once by a test suite. For example, in Example 1 (temperature and pressure sensors), achieving full statement coverage requires that each if branch in the validation logic is exercised, ensuring that both valid and invalid inputs are encountered at least once.
- *Branch coverage*: measures the proportion of decision branches executed. Each conditional in the program has multiple branches (true/false), and branch coverage ensures that every possible outcome of each conditional is exercised. In Example 1, branch coverage would require test cases where (i) both temperature and pressure are within valid ranges, (ii) temperature is invalid, (iii) pressure is invalid, and (iv) both are invalid, so that all if outcomes are covered.
- *Modified condition/decision coverage (MC/DC)*: evaluates the independent effect of each condition in a compound decision on the outcome of that decision. Using Example 1, MC/DC would require constructing a test suite where each condition (`validateTemperature(...)`, `validatePressure(...)`) independently affects the result of the if statement. For instance, one test could make only `validateTemperature` fail while `validatePressure` passes, and another could do the opposite, ensuring that each condition's impact is observed.
- *All-Uses coverage*: a data-flow criterion that tracks variable definitions (`def`) and their subsequent uses (`use`) across the program. It ensures that every def-use pair is exercised by at least one test case. For Example 2 (continuous vehicle speed), All-Uses coverage would require tests that propagate different speed readings from the sensor (`def`) through the acceleration validator and finally to the throttle adjustment logic (`use`), including cases with normal, maximum, and minimum accelerations.

These structural coverage criteria are widely employed in automated test generation and are particularly prevalent in safety-critical domains, including aerospace, automotive, and industrial control systems. MC/DC, in particular, is embedded in certification standards such as DO-178C and ISO 26262. By providing a clear and quantifiable framework, these criteria guide the systematic construction of test suites that exercise program logic comprehensively. At the same time, numerous empirical studies [6, 11, 19–23] indicate that the level of structural coverage alone is not a reliable indicator of test-suite effectiveness. Although such criteria guarantee the exercise of structural elements of a program, they do not directly account for the diversity of input values (and, consequently, their semantic significance) at the points of subsequent use. Such coverage may prove insufficient for safety-critical systems, in which it is necessary to test different parameter values governing actuator operation. The question of whether there exists a coverage criterion that most effectively guides the testing process toward defect detection remains open and controversial [20–24]. In particular, study [20] demonstrates that test data generated on the basis of structural criteria using counterexamples may be less effective than randomly generated tests. The authors concluded that even satisfying stringent criteria such as MC/DC does not guarantee a high fault-detection capability of the test suite. Similarly, the study presented in [21], devoted to the analysis of automatically generated tests in the avionics domain, emphasizes the need for reliable methods of evaluating test quality and for a more well-founded selection of testing coverage objectives.

Example 7. Consider the following code snippet:

```
01. void updateControl(mode, limit, alarm) {
02.     float d; // deviation from target flow rate
03.     if (mode == SENSOR_P)
04.         d = sensorP(); //Precise measurement
05.     else
06.         d = sensorE(); //Estimated (less reliable)
07.     float u = 2 * d;
08.     if(d<0 || d>limit || alarm)
09.         shutdownFlow();
10.     else
```

```

11.         applyControl(u);
12.}

```

In this example, the value of variable `d` depends on the data source and, consequently, may vary in accuracy. The decision to switch the system into safe mode is determined by a compound logical condition that combines exceeding a critical threshold with a contextual condition related to the alarm state (`alarm`). A test suite satisfying the Branch Coverage, MC/DC, and All-uses criteria is presented in Table 1.

Table 1

Test suite generated for Example 7

#	mode	d	limit	alarm	parameter of <code>applyControl</code>
T1	SENSOR_P	13	16	0	$0 \leq u \leq 32$
T2	SENSOR_E	5	30	1	no function call
T3	SENSOR_E	25	20	0	no function call
T4	SENSOR_P	-1	20	0	no function call

Indeed, the pair of tests T1 and T3 verifies the independent effect of the predicate ($d > \text{limit}$), the pair T1 and T4 verifies the condition $d < 0$, and the pair T1 and T2 verifies the effect of the variable `alarm`. For variable `d`, test T1 ensures coverage of the def-use pair 04–07, whereas tests T2, T3, and T4 cover the pair 06–07. For variable `u`, the only def-use pair 07–11 is covered exclusively by test T1, in which the value corresponds to sensor `SENSOR_P`.

Although both sensors employ the same value-processing structure, their operating ranges may differ substantially. Specifically, values obtained from `SENSOR_P` (*precise*) are characterized by relatively narrow and well-calibrated ranges, whereas `SENSOR_E` (*estimated*) is more sensitive to environmental influences. Although their ranges partially overlap, each sensor may also produce values that fall outside the typical range of the other. However, the achieved coverage does not guarantee that program execution encompasses the value ranges characteristic of each sensor modality, nor does it ensure the examination of values lying outside the range associated with an alternative definition. In particular, coverage of the definition–use pair for data originating from `SENSOR_P` does not imply that values specific to `SENSOR_E` have been explored.

Similarly, covering all branches of the validator in Example 1 does not guarantee that the downstream actuator (`adjustHeatingCooling`) experiences the full spectrum of plausible input variations, the same situation with the continuous throttle actuator in Example 2. These examples highlight the relevance of RS-DFC for testing cyber-physical systems, where correctness depends not only on reaching a particular computation point or controlling actuators, but also on the physical context encoded in the values flowing through the system. Unlike traditional approaches, RS-DFC enables distinguishing semantically different ranges of these values. Furthermore, it improves coverage observability — the ability of a coverage criterion to reflect the diversity of values influencing program behavior — and, consequently, enhances fault-detection capability.

3.2. Automated Test Generation

Automated test generation is widely used to reduce manual effort and improve the coverage of program logic. Existing approaches can be broadly categorized as follows:

- *Search-based test generation:* Tools such as EvoSuite employ evolutionary algorithms to generate test inputs that maximize a given structural coverage criterion. These tools iteratively evolve candidate test suites, selecting and mutating inputs that increase coverage. Search-based methods are effective at systematically exercising control- and data-flow paths, but they generally do not account for the semantic diversity of input values at sink locations. For CPS applications, this limitation may result in test suites that exercise all code branches yet fail to explore critical value ranges, such as specific sensor readings that trigger different actuator behaviors.

- *Random-based or fuzzing approaches:* Tools such as Randoop generate tests by randomly executing sequences of method calls and capturing outputs. Random testing is often effective for discovering unexpected program states, particularly in complex APIs, but coverage is typically incomplete and uneven, especially for inputs that must satisfy physical constraints or continuous ranges, as is common in CPS. While random approaches may occasionally hit boundary or invalid values, they do not systematically ensure that the range of physically meaningful input values is exercised.

- *AI-assisted test generation:* Recent advances in large language models (LLMs) have enabled generating test cases from natural-language prompts, code context, or documentation. LLMs can propose plausible inputs, edge cases, or invalid values based on learned patterns across software projects. Advantages include:

- Rapid generation of semantically meaningful test cases without requiring formal specifications.
- Ability to infer interactions between input variables, suggesting combinations that human engineers might overlook.
- Potential to produce test cases covering rare or unexpected scenarios, which can complement structural coverage criteria.

Limitations of LLM-based approaches include:

- Non-deterministic outputs: different model queries may produce different tests, making reproducibility and regression testing challenging.
- Lack of guarantees: unlike formal methods, LLMs do not systematically ensure coverage of all structural paths or input ranges.
- Potential for unrealistic inputs: generated tests may include values that are syntactically valid but physically impossible for the CPS under study.

None of these approaches alone guarantees that critical value ranges are fully exercised. For instance, in Example 1 (temperature and pressure), structural coverage-focused test generation may cover all branches of the `if` statements, but miss edge values near sensor limits or combinations of noisy readings. Similarly, in Example 2 (continuous acceleration), random or LLM-generated tests may fail to systematically cover the entire spectrum of valid accelerations and corresponding actuator responses.

A variety of other methods have been proposed for automated testing of software and CPS. While these approaches are certainly valuable in specific contexts, they generally operate orthogonally to range-sensitive dataflow coverage: many require extensive environment modeling, focus primarily on robustness or high-level system behaviors rather than systematic coverage of input value ranges, rely on formal specifications, or are restricted to low-level firmware testing. Symbolic and concolic testing are included among these approaches, but by themselves they are not specifically designed to achieve extended dataflow coverage of input constraints. Therefore, these methods are best viewed as complementary, providing different perspectives on testing rather than directly ensuring coverage of input constraints at downstream use points. Examples of such methods, along with representative tools and their limitations, are summarized in Table 2.

Table 2

Summary of representative methods

Category	Principle	Open-Source Examples	Limitations / Applicability
Fuzzing / Coverage-Guided Input Generation	Automatic generation of diverse inputs to trigger unexpected system behavior; coverage-guided selects new tests based on coverage	AFL / AFL++; OneFuzz; ICS3Fuzzer	Works well for individual components or communication interfaces; not inherently CPS-aware; limited modeling of physical environment

Category	Principle	Open-Source Examples	Limitations / Applicability
Simulation-Based Scenario Testing	Generation of scenarios in CPS models (sensor/actuator inputs, environment perturbations)	ROS + Gazebo; BeamNG (SBFT benchmarks)	Requires environment modeling; difficult to scale to all input spaces; typically for high-level system behavior
Symbolic Concolic Testing	Generates inputs by analyzing code paths and branch conditions; aims to cover all logical paths	KLEE; angr (binary-level concolic testing)	Poor scalability for complex CPS; mostly suited for modules with critical input validation logic
Fault Injection (Runtime Sensor Network)	Injects faults into system inputs at runtime, including sensors or network messages	FIBlock (Simulink); FIESTA framework; FAIL*	Requires simulation or access to embedded environment; focuses on robustness rather than formal input validation and coverage
Embedded Firmware Input Testing	Targets low-level handling of inputs in CPS/IoT firmware	FIRM-AFL	Limited to firmware/binary-level testing; requires emulation or instrumentation

Summary. The limitations of existing approaches motivate range-sensitive, context-aware testing, which explicitly tracks input-originated values from definitions to downstream uses and ensures that test suites exercise distinct equivalence classes of values under realistic operating conditions. Such an approach can complement traditional automated test generation tools by focusing on semantically relevant input diversity, rather than purely structural code paths.

4. Range-Sensitive Dataflow Coverage

This section introduces fundamental dataflow concepts that underpin our analysis, including definition–use associations and reaching definitions. These notions are used to define and reason about the proposed coverage criterion [13].

Definition 4.1 (Program Model). Let a program be represented as a finite directed graph $G=(V,E,\ell_0)$, where each node $\ell \in V$ corresponds to a single statement, $E \subseteq V \times V$ is the control-flow edges, and path is a sequence $\pi = \langle \ell_0, \ell_1, \dots \rangle$, where $\ell_i \in V$ and $(\ell_i, \ell_{i+1}) \in E$. Statements are classified as guards (conditional decisions) or computations (all other statements). For a variable x , let $\text{Def}(x)$ and $\text{Use}(x)$ denote its definition and uses nodes, with $\text{PUse}(x) \subseteq \text{Use}(x)$ for uses in guard predicates, and $\text{CUse}(x) \subseteq \text{Use}(x)$ for all other computation uses.

A definition of a variable occurs at a program statement where a value is assigned to the variable. Typical cases include direct assignments, initialization via formal parameters, or updates through arithmetic or compound assignment operations. Conversely, a use of a variable occurs at a statement where the current value of the variable is read. Common examples include evaluation in conditional expressions, use in the right-hand side of assignments, or passing the variable as an argument to a function.

Definition 4.2 (Reaching Definition). Let $d \in \text{Def}(x)$ and $u \in \text{Use}(x)$. The definition d of variable x is said to reach the use u , iff: there exists a path $\pi = \langle d, \dots, u \rangle$ in G , and no node on π redefines x (π is a “def-clear” path with respect to x [10]).

A definition–use association for a variable v is a pair (d, u) such that the definition d reaches the use u . In the remainder of the paper we refer to such associations either as def–use pairs or, when emphasizing the variable, as the tuple (d, u, v) .

Let us consider a program P with a set of variables V and let $\text{DU}(P)$ denote the set of all def–use pairs in P . Traditional All-Uses coverage requires that for each $(d, u, v) \in \text{DU}(P)$, there exists at least

one test case t such that the execution of P with input t reaches d and subsequently u via def-clear path [10] with respect to the variable v . While this ensures that every definition reaches all its uses, it does not distinguish between different values that may flow from the definition to the use [11]. The range-sensitive dataflow coverage criterion strengthens All-Uses coverage by requiring that all meaningful value partitions at each use location are exercised.

Definition 4.3 (Domain Ranges). Let v be a variable over an ordered domain D . A range of v is a subset $R \subseteq D$ of values satisfying a given predicate $\phi(v)$. For ordered domains (e.g., integers, floating-point numbers), we define ranges as maximal contiguous subsets, that is, $R = \{x \in D \mid \phi(x)\}$ such that for any $x, y \in R$ and any $z \in D$ with $x < z < y$, we have $z \in R$. For unordered or discrete domains (e.g., booleans, enums), ranges may be singleton values or non-contiguous subsets. In either case, let $R_{v,u} = \{r_1, r_2, \dots, r_k\}$ denote a finite set of ranges under consideration for testing (or abstract equivalence classes) that partition all values of v that can reach use location u .

Example: For $v \in 0..10 \wedge v \neq 5$, the ordered domain produces two ranges: $0 \leq v < 5$ and $5 < v \leq 10$. For a Boolean variable, the two singleton ranges are $\{\text{true}\}$ and $\{\text{false}\}$.

The ranges $R_{v,u}$ can be determined statically (from type information, specifications, assertions, or domain knowledge) or dynamically (generated on-the-fly during symbolic execution as new constraints are discovered). A test case t covers a range $r \in R_v$, if, during execution with input t , the value of v at u lies within r .

Definition 4.4 (Range-Sensitive Dataflow Coverage, RS-DFC). A test suite $T = \{t_1, t_2, \dots, t_n\}$ satisfies RS-DFC for program P if and only if for every def-use pair $(d, u, v) \in DU(P)$ and for every range $r_i \in R_{v,u}$, there exists at least one test $t_j \in T$ such that v takes a value in r_i at u during execution of P with input t_j . Formally:

$$(d, u, v) \in DU(P) \wedge r_i \in R_{v,u} \Rightarrow \exists t_j \in T: v(t_j, u) \in r_i$$

where $v(t_j, u)$ denotes the value of variable v at statement u during execution of t_j .

In this study, we focus RS-DFC coverage specifically on sink-oriented def-use pairs, i.e., pairs where the use corresponds to a program location that directly consumes a variable in a way that may influence observable system behavior, actuator outputs, or interactions with the environment. Def-use pairs whose use does not affect such sinks are excluded from coverage obligations.

Definition 4.5 (Completeness of a RS-DFC Test Suite). A test suite T is said to be complete with respect to a variable v and a sink location s if it exercises all feasible ranges of v that can reach s . Let FeasibleTraces denote the set of all feasible abstract traces of the program. Then T is complete if, for every feasible trace $t' \in \text{FeasibleTraces}$, there exists a test $t \in T$ such that the range of v at s in t' is entirely covered by the range observed in t :

$$\forall t' \in \text{FeasibleTraces}, \exists t \in T: (v, s, t') \subseteq R(v, s, t)$$

Here (v, s, t) denotes the range of values of v observed at sink s during execution of test t . This definition formalizes range-sensitive completeness: a complete RS-DFC test suite guarantees that all feasible input-dependent values reaching sink locations are exercised by at least one test, supporting effective detection of subtle faults in CPS input validation logic.

Feasibility. The criterion has a natural interpretation in terms of feasibility over explicitly defined ranges. Given a finite partitioning $R_{v,u}$, coverage can be evaluated by executing the test suite and checking whether the value of variable v at use location u falls within each specified range. Ranges that are infeasible (i.e., cannot be realized by any execution) are treated analogously to infeasible def-use pairs in classical dataflow testing: they are simply regarded as uncovered and do not invalidate the definition of the criterion.

Scope and applicability. The proposed criterion is not uniformly beneficial for all definitions and uses. Its effectiveness depends on the presence of meaningful variability in the values propagated from definitions to their corresponding uses. In particular, definitions that assign constant values do not induce variability: the set of values reaching any use is a singleton. In such cases, distinguishing between value ranges does not provide additional testing insight beyond traditional criteria. For this

reason, the criterion is primarily intended for input-dependent definitions, whose values are determined by external inputs, sensor readings, environmental interactions, or computations over previously unconstrained data. These definitions may produce a broad (potentially unbounded) set of values, making them suitable for partitioning into representative ranges. A similar consideration applies to use locations. If the value of a variable is restricted to a single feasible value by dominating conditions prior to a use, then range-based distinctions at that use are not meaningful. In such cases, RS-DFS would not contribute beyond the All-Uses criterion.

Among all def–use pairs, particular attention is given to definition–computation-use (Def–CUse) pairs. Computation uses act as sink points in the dataflow, where variable values directly influence program behavior through arithmetic operations, external interactions, or state updates. These locations are often more prone to subtle faults, especially in CPS contexts where they may affect physical processes. In contrast, predicate uses (PUse) primarily influence control flow and are already effectively exercised by criteria such as branch coverage or MC/DC; therefore, they typically do not benefit from additional range sensitivity.

Example. Being applied to the example 7, the coverage criterion will require the additional test cases presented in Table 3.

Table 3

Additional test cases required by RS-DFC

#	mode	d	limit	alarm	parameter <code>applyControl(u)</code>
T5	SENSOR_P	D1	L1	0	$0 \leq u \leq 2 \cdot L1$
T6	SENSOR_E	D2	L2	0	$0 \leq u \leq 2 \cdot L2$

Here values of $D1, L1 \in D(\text{SENSOR_P}) \setminus D(\text{SENSOR_E})$, $D1 < L1$, and similarly $D2, L2 \in D(\text{SENSOR_E}) \setminus D(\text{SENSOR_P})$, $D2 > L2$. That is, coverage of values that are unique to different sensors is required.

5. Benchmark and Mutation Model

5.1. Benchmark Programs

To evaluate the proposed range-sensitive coverage perspective, we constructed a benchmark suite of programs [14] modeling input constraint enforcement typical of CPS. The benchmark focuses on code fragments where externally supplied values—such as sensor readings—are validated and subsequently used in computations affecting system behavior. The programs are derived from representative validation patterns observed in CPS software. These include checks for admissible ranges of physical quantities, consistency constraints across multiple sensors, rate-of-change limits reflecting physical dynamics, and context-dependent constraints conditioned on operational modes. Each benchmark program contains one or more sink locations where validated values influence program behavior.

The design of the benchmark is guided by the CPS-specific challenges outlined in Section 2. In particular, the programs capture characteristics such as heterogeneous input sources, continuous and real-time behavior, safety-critical decision logic, interactions between multiple inputs, and context-dependent constraints. Rather than modeling complete systems, the benchmark isolates these aspects in small, self-contained programs, allowing their impact on testing effectiveness to be studied in a controlled manner.

To support systematic evaluation, the benchmark suite is intentionally lightweight: individual programs are relatively small (typically 20–50 lines of code) and focus on a limited number of constraints and variables. This design enables clear identification of induced value partitions and facilitates comparison of different test generation approaches in terms of how thoroughly they explore value ranges reaching sink locations.

Table 4 summarizes the main characteristics of the benchmark programs, including their CPS domain, validation patterns, and the specific CPS-related challenges they represent.

Table 4

Characteristics of the benchmark suite

Source	CPS Domain	Validation Pattern	#Inputs	#Constraints
ApolloAuto	Automotive / Autonomous Driving	Sensor fusion + trajectory consistency	4	2–5
ArduPilot	Flight Control / Robotics	Redundant sensors + fail-safe	5	6
AutoWareFoundation	Automotive / Autonomous Driving	Real-time perception consistency	6–7	2–3
JSBSim	Flight Dynamics / Simulation	Non-linear interaction	4–10	7–17
openETCS	Railway Control	Mode-dependent constraints	6	9
OpenPilot	Automotive / Driver Assistance	Rate-of-change (steering/braking)	3	2–5
OpenVSP	Aerospace Design / Simulation	Geometric consistency constraints	3	5–13
PX4-Autopilot	Flight Control / Robotics	State estimation + fail-safe	9–10	4–5

5.2. Mutation Operators for Validation Faults

To evaluate the fault-detection capability of generated test suites, we employ mutation analysis by introducing a set of operators that model realistic validation faults. Each operator represents a small syntactic modification of the program, simulating typical implementation mistakes in input constraint enforcement.

Mutation analysis [25] is grounded in two widely accepted assumptions: the Competent Programmer Hypothesis and the Coupling Effect. The former assumes that real programs are typically close to correct implementations, while the latter states that test cases capable of detecting simple faults are also effective at revealing more complex ones. Based on these assumptions, mutation operators are designed to introduce small, localized changes, each representing a single fault, without altering the overall structure of the program.

In the context of CPS, validation logic is particularly prone to errors involving incorrect thresholds, missing checks, inconsistent conditions, and improper handling of sensor values. To capture these patterns, we define a set of mutation operators that combine general-purpose fault models with validation-specific and CPS-oriented modifications.

Table 5

Mutation operators for validation faults

Operator	Description	Example
Boundary Shift	Off-by-one or threshold errors	$x \leq 100 \rightarrow x < 100$
Range Expansion	Accepting invalid values	$x \leq 100 \rightarrow x \leq 200$
Range Contraction	Rejecting valid values	$x \leq 100 \rightarrow x \leq 80$
Relational Operator Replacement	Incorrect comparison	$x < 100 \rightarrow x > 100$
Arithmetic Operator Replacement	Incorrect arithmetic	$x + y \rightarrow x - y$
Condition Negation	Inverting validation logic	$\text{if}(\text{valid}(x)) \rightarrow \text{if}(!\text{valid}(x))$
Missing Validation	Removing validation check	remove $\text{if}(x < 0)$

Operator	Description	Example
Missing Condition	Dropping part of predicate	$\text{if}(a \ \&\& \ b) \rightarrow \text{if}(a)$
Variable Substitution	Wrong variable used	$\text{if}(\text{temp} < 100) \rightarrow \text{if}(\text{pressure} < 100)$
Constant Mutation	Incorrect threshold value	$100 \rightarrow 1000$
Sensor Stuck-at Value	Sensor returns constant	$x = \text{read}() \rightarrow x = 0$

Table 6
CPS-specific mutation operators

Operator	Description	Example
Rate Constraint Fault	Incorrect rate-of-change limit (continuous dynamics)	$\text{fabs}((v2 - v1)/dt) \rightarrow \text{fabs}(v2 - v1)$
Rate Computation Fault	Incorrect difference computation	$\text{fabs}((v2 - v1)/dt) \rightarrow (v2 - v1)/dt$
Consistency Check Fault	Weakening relation between redundant sensors	$\text{fabs}(\text{temp1} - \text{temp2}) < 5 \rightarrow < 50$
Context Condition Fault	Wrong constraint applied in a mode	$\text{case NORMAL: } x \leq \text{MIN} \rightarrow x \leq \text{MAX}$
Mode Switch Fault	Incorrect branch selection	$\text{if}(\text{mode} == \text{NORMAL}) \rightarrow \text{if}(\text{mode} == \text{EMERGENCY})$
Temporal Fault	inconsistent time samples	$v_{\text{prev}} \rightarrow v_{\text{prev_old}}$
Time-based validation fault / Staleness check fault	Incorrect temporal validation logic / Timestamp validation fault	$\text{if}(\text{curr_time} - \text{sensor.timestamp} < \text{MAX_AGE}) \rightarrow \text{if}(\text{sensor.timestamp} < \text{MAX_AGE})$
Block Scope Fault	Misplaced validation scope (dominance frontier fault)	<pre> if(x > 0) { validate(x); } use(x); instead of guarding both </pre>
Delayed Validation Fault	Validation applied too late	<pre> use(x); validate(x); (swapped order) </pre>

The first group (Table 5) of operators corresponds to classical mutation categories, including arithmetic, relational, and logical faults, targeting validation-specific errors that directly affect the enforcement of input constraints. These mutations represent common programming mistakes, such as incorrect operators, missing conditions, or wrong constants, which can lead to missing validation, incorrect thresholds, or incomplete compound predicates. Such faults are particularly relevant for testing input-dependent logic and assessing the effectiveness of range-sensitive coverage.

The second group (Table 6) captures CPS-specific fault patterns, which arise from the interaction between software and physical processes. These include faults in rate-of-change constraints, inconsistencies between redundant sensors, incorrect handling of context-dependent constraints, and faults affecting the scope of validation logic. In particular, block scope faults (e.g., misplaced braces) may unintentionally expand or restrict the region where validation is applied, leading to subtle errors in downstream behavior.

A block scope fault occurs when the intended scope of a validation check is altered due to incorrect placement of control-flow boundaries, such as misplaced or missing braces. As a result, validation may apply only to a subset of statements, while subsequent operations unintentionally execute on unvalidated inputs. In CPS, this can lead to situations where safety checks appear present in the code but do not effectively guard all relevant computations or actuator commands. Such faults are particularly subtle, as they do not remove validation logic but weaken its enforcement. Structural coverage criteria typically fail to expose this issue, since the validation condition is still executed. In

contrast, analyzing the values that reach sink locations can reveal that unvalidated or out-of-range inputs propagate to critical operations.

```
// Block Scope Fault Example: incorrect closing of conditional block
float newValue = readSensor(); // assume some sensor reading
// Original intended validation
if (fabs(prevValue - newValue) < Delta) {
    // value is valid, proceed
    prevValue = newValue;
    activateActuator(newValue);
} // end of valid block
// Mutant: closing brace misplaced
if (fabs(prevValue - newValue) < Delta) {
    // value is valid, proceed
    prevValue = newValue;
} // end of valid block
// code block executes unconditionally
activateActuator(newValue);
```

This snippet illustrates an example where a misplaced closing brace causes the validated block to become ineffective. In the mutant, the `activateActuator` function is executed regardless of input validity, potentially causing unsafe actuator commands.

A *delayed validation fault* occurs when input values are used in downstream computations before validation checks are applied. This type of fault reflects incorrect ordering between validation and usage, allowing invalid or corrupted inputs to influence system behavior. In CPS, such errors are particularly critical, as they may lead to unsafe actuator commands before the violation is detected. From a testing perspective, these faults are difficult to detect using structural coverage alone, as the validation logic is still executed. However, they can be exposed by analyzing the ranges of values that reach sink locations.

```
float speed = readSensor();
// Downstream usage BEFORE validation
applyBrakingForce(speed);
// Validation happens too late
if (speed < 0 || speed > 200) {
    return ERROR;
}
```

5.3. Mutation Execution and Killing Criterion

In this work, we apply “strong mutation” criterion, where the effect of a mutant is evaluated based on observable differences at designated sink locations. A test case is said to kill a mutant if, when executed on the original and mutated program, it produces a difference in at least one of the following aspects:

- whether a sink function is invoked, or
- the values of one or more arguments passed to a sink function.

Sink functions represent program points where validated inputs are ultimately used, such as control actions, actuator commands, or externally visible computations. By focusing on these locations, the evaluation captures the semantic impact of input validation and propagation, rather than intermediate internal states.

During evaluation, a mutant is considered killed by a test suite if at least one test case produces a difference in observable behavior at sink locations (e.g., a sink function is called differently or with altered parameter values) compared to the original program. Mutation scores are then computed as the ratio of killed mutants to the total number of generated mutants.

This definition differs from traditional weak mutation, which checks for state differences immediately after the mutated statement. Instead, the proposed criterion observes whether such

differences propagate to downstream use points, reflecting the practical effect of validation faults on system behavior. This approach is particularly suitable for CPS-inspired programs, where the correctness of input handling is determined by how values influence subsequent computations and control decisions.

5.4. Mutant Generation

For each benchmark program, mutants were generated using our prototype mutation tool by systematically applying both classical and CPS-specific mutation operators (Tables 5 and 6). Mutants that produced compilation errors or were equivalent to the original program were discarded. The resulting set of mutants represents potential validation faults that test suites should ideally detect.

The benchmark programs, the prototype mutation generation tool, and all generated mutants are publicly available for reproducibility and further research at [14].

6. Experimental Setup

6.1. Research Questions

This study evaluates the effectiveness of range-sensitive data-flow coverage (RS-DFC) for improving the exploration of input validation logic in cyber-physical system software. The focus is on whether structurally adequate test suites sufficiently exercise semantically distinct value ranges reaching sink locations, and whether additional range-sensitive testing improves fault detection.

Accordingly, the following research questions are addressed:

RQ1. Coverage enrichment. To what extent does RS-DFC require additional test cases beyond structurally adequate test suites generated under branch and MC/DC coverage constraints?

RQ2. Fault detection effectiveness. Does extending structurally adequate test suites with RS-DFC-relevant test cases improve mutation-based fault detection for input validation logic?

6.2. Subject Programs and Test Generation Approaches

The evaluation is conducted on a benchmark set of C++ programs implementing sensor input validation logic typical for cyber-physical systems. Each program includes validation constraints on sensor inputs and downstream usage of validated values in control or computation logic (sink operations).

To obtain structurally adequate baseline test suites, we use CBMC tool (Bounded Model Checker for C/C++ [26]), configured to generate test inputs achieving high branch and MC/DC coverage. These test suites serve as the structural baseline. For the All-Uses criterion, test suites were generated using our custom data-flow test generation framework [11, 13].

To evaluate the effectiveness of modern automated test generation, we additionally include tests generated by large language models. Two independent models are used to ensure robustness of results. For each model, test cases are generated using a unified prompt and the same test budget.

In addition, we include test cases generated by a large language model (LLM), specifically *GPT-5* and *Claude Sonnet 4.6*, to represent recent AI-assisted test generation techniques. LLM Test Generation Prompt (used for both models):

```
You are a software testing assistant. For the given C++ code,
generate exactly N unit tests. Each test must only call the function
"entry" with concrete input values to achieve diverse and meaningful
coverage of the function.
```

Here, N is defined as the total number of tests in the baseline CBMC-generated suite plus the additional tests required to satisfy RS-DFC coverage. This ensures equal test budget across all compared approaches.

6.3. Test Generation Protocol

For each benchmark program, experiments are conducted in three configurations:

- Baseline configuration: CBMC-generated test suites achieving branch and MC/DC coverage.

- RS-DFC augmented configuration: baseline suites extended with additional test cases required to satisfy RS-DFC obligations.
- LLM-generated configuration: test suites generated independently by two LLMs using the prompt above, with a fixed budget N .

All test suites are executed on the original programs and evaluated under identical conditions. No manual tuning of test cases is performed.

6.4. Evaluation Metrics

The following metrics are used:

- Test suite size increase: relative growth in the number of test cases after RS-DFC augmentation;
- Mutation score: ratio of killed mutants to total mutants, where mutants simulate typical validation faults in sensor processing logic;
- Structural adequacy (baseline): branch and MC/DC coverage provided by CBMC-generated tests (used as a precondition, not an optimization objective).

Together, these metrics allow comparison of structural adequacy, range-sensitive coverage, and learning-based test generation under a unified evaluation framework.

7. Results and Discussion

RS-DFC Enrichment of Structurally Adequate Test Suites (RQ1). The experimental results show that test suites generated to satisfy branch and MC/DC coverage do not fully satisfy RS-DFC requirements. In all benchmark programs, additional test cases were required to cover missing range-sensitive data-flow obligations at sink locations. On average, RS-DFC augmentation increased the size of the test suites by approximately 30%, indicating that structural coverage alone is insufficient to ensure full exploration of semantically distinct value ranges in input validation logic.

Impact on Fault Detection (RQ2). Extending structurally adequate test suites with RS-DFC-oriented test cases leads to a consistent improvement in mutation detection effectiveness. Across all benchmark programs, the augmented test suites achieved an average increase of approximately 15% in mutation score compared to the baseline CBMC-generated suites. This indicates that additional RS-DFC-driven test inputs target fault-revealing value regions that are not exercised by structurally guided test generation alone.

LLM-generated test suites show competitive performance in some cases; however, their effectiveness varies across programs and does not consistently match the improvements achieved by RS-DFC augmentation under a fixed test budget.

Discussion. The results show that high structural coverage does not necessarily imply sufficient exploration of validation-induced value regions. Test suites achieving strong branch, MC/DC, or All-Uses coverage may still leave significant input partitions at sink locations unexplored.

RS-DFC addresses this limitation by explicitly targeting value ranges propagated to sink operations. Augmenting structurally adequate test suites with RS-DFC-relevant tests increases mutation score from 69.73% (MC/DC baseline) to 84.91%, indicating that the additional tests exercise fault-relevant behaviors missed by structural criteria alone. LLM-generated test suites produce competitive results under the same test budget, but their effectiveness varies across programs and remains less consistent than RS-DFC-guided augmentation. Table 7 summarizes the average results across all benchmark programs.

Table 7

Average results across all benchmark programs

Metric	Branch Coverage	MC/DC	All-uses	GPT-5.6	Claude 4.6	RS-DFC
Avg # Tests	5.63	9.21	9.84	11.88	11.88	MC/DC +2.67
Avg Mutation Score	66.00%	69.73%	68.33%	79.73%	78.41%	84.91%

8. Threats to Validity

Construct Validity. Construct validity concerns whether the selected metrics adequately reflect test effectiveness for input validation logic.

Structural metrics such as branch coverage, MC/DC, and All-Uses capture exploration of program structure but do not directly measure coverage of validation-induced value regions. RS-DFC addresses this limitation by explicitly tracking value ranges reaching sink locations; however, the current formulation does not model interactions among multiple correlated variables.

Mutation score depends on the selected mutation operators and may not represent all real-world validation faults. To reduce this threat, mutation operators were designed to model representative validation errors commonly observed in sensor-processing logic.

Internal Validity. Internal validity concerns whether the observed effects are caused by the evaluated techniques rather than uncontrolled factors.

LLM-generated tests may vary across executions due to nondeterministic generation. To mitigate this threat, experiments were repeated multiple times and average results were reported.

In addition, RS-DFC augmentation is derived from the same coverage framework used during evaluation, which may favor RS-DFC-oriented test suites. To reduce this bias, all approaches were evaluated using the same mutation-based effectiveness criterion.

External Validity. The benchmark programs are intentionally small and focused on representative validation patterns found in cyber-physical systems. While this enables controlled evaluation, the results may not fully generalize to large-scale industrial software.

The study focuses on C/C++ programs and sensor-validation scenarios; different programming languages, runtime environments, or application domains may exhibit different behavior.

Conclusion Validity. The number of benchmark programs is relatively limited, reducing statistical power. However, the study is intended as a controlled exploratory evaluation focused on identifying observable effects and trends rather than establishing broad statistical generalization.

Mitigation Strategies. To support reproducibility and reduce experimental bias, we:

- designed benchmark programs and mutation operators based on representative validation patterns;
- repeated experiments across multiple runs;
- used identical test budgets for all compared approaches;
- provide benchmark programs, mutants, and experimental scripts as a replication package.

9. Conclusion

The experimental results obtained on the benchmark suite of sensor-data validation programs demonstrate that traditional structural coverage criteria, particularly branch coverage and MC/DC, may create a misleading impression of testing completeness. Even structurally adequate test suites often fail to explore critical value ranges relevant to validation logic.

The proposed RS-DFC criterion extends traditional data-flow coverage by explicitly distinguishing value ranges propagated to sink locations. Experimental evaluation shows that augmenting structurally adequate test suites with RS-DFC-relevant tests improves mutation detection capability by approximately 15%, while requiring a moderate increase in test-suite size of approximately 30% compared with MC/DC and All-Uses baselines. These results indicate that coverage of validation-induced value regions contributes additional fault-detection capability beyond conventional structural metrics.

The findings are particularly relevant for cyber-physical systems, where validation logic determines whether sensor inputs influence downstream control decisions or actuator behavior. Insufficient exploration of value regions may therefore leave safety-critical behaviors inadequately tested.

Experimental artifacts, including the mutation testing framework for cyber-physical systems, the mutant generator, and the benchmark test suites, are publicly available in the open repository [14].

Future work includes extending the benchmark suite to larger-scale cyber-physical systems, modeling interactions among multiple validated variables, and developing automated techniques for guiding test generation toward uncovered value regions.

Declaration on Generative AI

ChatGPT 5.6 was used for text translation and language polishing. The authors reviewed and edited the content and take full responsibility for the publication's content.

References

- [1] A. Letichevsky, O. Letychevskiy, V. Skobelev, V. Volkov, Cyber-physical systems. *Cybernetics and Systems Analysis*. 2017. Vol.53(6), pp. 821–834. doi:10.1007/s10559-017-9984-9.
- [2] X. Zhou, X. Gou, T. Huang, S. Yang, Review on testing of cyber physical systems: Methods and testbeds. *IEEE access*. 2018. Vol. 6. 52179–52194. doi:10.1109/ACCESS.2018.2869834.
- [3] C. Berger, M. Rumpe, B. Dörr, C. Schmid, Characterization of cyber-physical sensor systems. *Procedia Cirp*. 2016. Vol.41. pp. 638–643. doi:10.1016/j.procir.2015.12.019.
- [4] E. Kurian, D. Briola, P. Braione, G. Denaro, Automatically generating test cases for safety-critical software via symbolic execution. *Journal of Systems and Software*. 2023. 111629. doi:10.1016/j.jss.2023.111629.
- [5] B. Homes, *Fundamentals of software testing*, ISTE Ltd, (book). 2024. 1–338. doi:10.1002/9781394298976.
- [6] V. Volkov, A. Kolchin, A. Letychevskij, S. Potienko, A survey of systematic methods for code-based test data generation. *Artificial Intelligence*. 2017. Vol. 2. 71–84. URL:www.jnas.nbu.gov.ua/article/UJRN-0000864496.
- [7] Z. Sadri-Moshkenani, J. Bradley, G. Rothermel, Survey on test case generation, selection and prioritization for cyber-physical systems, *Software Testing, Verification and Reliability* 32 (2022) e1794. doi:10.1002/stvr.1794.
- [8] J. Chilenski, S. Miller, Applicability of modified condition/decision coverage to software testing, *Software Engineering Journal* 9(5) (1994) 193–200. doi:10.1049/sej.1994.0025.
- [9] T. Su, K. Wu, W. Miao, G. Pu, J. He, Y. Chen, Z. Su, A survey on data-flow testing, *ACM Computing Surveys* 50(1) (2018) Article 1, 35 pages. doi:10.1145/3020266.
- [10] S. Rapps, E. Weyuker, Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on software engineering* SE-11(4) (1985) 367–375. doi:10.1109/TSE.1985.232226.
- [11] A. Kolchin, S. Potiyenko, Extending Data Flow Coverage to Test Constraint Refinements. In: ter Beek, M.H., Monahan, R. (eds) *Integrated Formal Methods. IFM 2022. Lecture Notes in Computer Science*, vol 13274, 2022. doi:10.1007/978-3-031-07727-2_17.
- [12] O. Kolchyn, D. Rudenko, A Formal Constraint-Based Method for Detecting Gaps in Input Validation Test Suites. In: Gervasi, O., et al. *Computational Science and Its Applications – ICCSA 2026, Lecture Notes in Computer Science*, vol 16768. Springer, Cham, 2026, 368–380. doi:10.1007/978-3-032-30488-9_23.
- [13] O. Kolchyn, D. Rudenko, Model-Based Test Cases Generation for Range-Sensitive Dataflow Coverage. *Cybernetics and Computer Technologies*. 2026. No 2. 103–118. doi:10.34229/2707-451X.26.2.9.
- [14] D. Rudenko, Toolset for Experimental Mutation Testing of Cyber-Physical Systems. GitHub rep., 2026. [Online]. URL:https://gitlab.com/DashaRudenko/cpf_experiments1.
- [15] Z. Yang, R. Huang, C. Cui, N. Niu, D. Towey, Requirements-based test generation: A comprehensive survey. *ACM Transactions on Software Engineering and Methodology*. 2025. doi:10.1145/3771727.

- [16] A. Herrera, M. Payer, A. Hosking, DatAFLOW: Toward a Data-Flow-Guided Fuzzer, *ACM Transactions on Software Engineering and Methodology*. 2023. Vol. 32(5). 31 p. doi:10.1145/3587156.
- [17] M. Kooli, F. Kaddachi, G. Natale, A. Bosio, P. Benoit, L. Torres, Computing reliability: On the differences between software testing and software fault injection techniques. *Microprocessors and Microsystems*. 2017. 50:102–112. doi:10.1016/j.micpro.2017.02.007.
- [18] G. Vidyalakshmi, S. Gopikrishnan, W. Boulila, A. Koubaa, Digital twins and cyber physical systems a new frontier in computer modeling. *Computer Modeling in Engineering and Sciences*. 2025. 143:51–113. doi:10.32604/cmescs.2025.057788.
- [19] M. Staats, G. Gay, M. W. Whalen, M. Heimdahl, On the danger of coverage directed test case generation. In: de Lara, J., Zisman, A. (eds) *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, vol. 7212. 2012. 409–424. doi:10.1007/978-3-642-28872-2_28.
- [20] M. Heimdahl, M. Whalen, A. Rajan, M. Staats, On MC/DC and implementation structure: An empirical study. In *Digital Avionics Systems Conference*, 2008, pp. 5.B.3-1–5.B.3-13. doi:10.1109/DASC.2008.4702848.
- [21] G. Gay, M. Staats, M. Whalen, M. Heimdahl, The risks of coverage-directed test case generation. *IEEE Transactions on Software Engineering* 41(8), 2015. pp. 803–819. doi:10.1109/TSE.2015.2421011.
- [22] G. Gay, A. Rajan, M. Staats, M. Whalen, M. P. E. Heimdahl, The effect of program and model structure on the effectiveness of MC/DC test adequacy coverage. *ACM Transactions on Software Engineering and Methodology*. 2016. 25(3), 1–34. doi:10.1145/2934672.
- [23] L. Inozemtseva, R. Holmes, Coverage is not strongly correlated with test suite effectiveness. *36th International Conference on Software Engineering, ICSE 2014*, ACM, 2014, pp. 435–445. doi:10.1145/2568225.2568271.
- [24] S. Al Blwi, A. Ayad, B. Khairiddine, I. Marsit, A. Mili, Semantic Coverage: Measuring Test Suite Effectiveness. *18th International Conference on Software Technologies, ICSOFT 2023*, 2023, pp. 287–294. doi:10.5220/0012063900003538.
- [25] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, M. Harman, Mutation testing advances: an analysis and survey. *Advances in computers*. 2019. Vol. 112. Elsevier. pp. 275–378. doi:10.1016/bs.adcom.2018.03.015.
- [26] D. Kroening, P. Schrammel, M. Tautschnig, CBMC: The C Bounded Model Checker, *arXiv preprint arXiv:2302.02384* (2023). doi:10.48550/arXiv.2302.02384.