

A Formal Constraint-Based Method for Detecting Gaps in Input Validation Test Suites

Oleksandr Kolchyn(✉) and Daria Rudenko

V.M. Glushkov Institute of Cybernetics NAS of Ukraine, Kyiv, Ukraine
kolchin_av@yahoo.com

Abstract. Insufficient input validation remains a common source of software vulnerabilities and functional errors. Although numerous structural coverage criteria exist in software testing, they generally do not specifically target the completeness or inconsistencies of input validation itself. In this paper, we propose a formal, specification-agnostic method for systematically identifying gaps in existing test suites with respect to input validation logic. It tracks symbolic constraints on tainted variables from source to sink locations along feasible execution paths and analyzes their subsumption relations to identify test cases that reveal previously untested validation scenarios. Our method can therefore reveal unduly weak or overly restrictive input constraints, thereby providing insights into potential defects in the implemented validation checks.

Keywords: testing · coverage criteria · input validation · symbolic constraints

1 Introduction

Thorough input validation is an essential defense against both accidental faults and malicious data. Numerous studies, including the Common Weakness Enumeration (CWE) rankings [1], identify weak or inconsistent input validation as one of the most frequent sources of software vulnerabilities. Despite its critical role, it is still difficult to systematically assess how adequate input validation testing is.

Requirements-based testing alone may miss important validation paths, especially when specifications are incomplete or imprecise. For this reason, practitioners usually combine it with structural white-box criteria to achieve broader coverage [2]. This combination improves path exploration, but it does not directly measure how thoroughly the validation logic itself is exercised. Even if all tests cover the documented requirements and meet strict coverage criteria [2–5], some aspects of validation may remain untested. For example, a test can appear redundant under strong criteria such as MC/DC [6] or all-uses [7], yet still provide useful insight by exercising insufficiently tested input constraints or exposing inconsistencies in the validation logic [3, 4, 8, 9].

To cover different behaviors while reducing redundancy, testers often use equivalence class testing [2, 10]. This approach partitions the input domain into subsets that are

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2027
O. Gervasi et al. (Eds.): ICCSA 2026, LNCS 16768, pp. 368–380, 2027.
https://doi.org/10.1007/978-3-032-30488-9_23

expected to produce similar behavior and then selects representative test cases. However, such partitions rely on assumptions made by the tester and may not reflect the actual validation logic implemented in the code. As a result, some validation conditions can be overlooked, leading to incomplete coverage.

The effect of input validation lies in how the program later consumes the validated data. Therefore, equivalence class testing can be applied more precisely at sink locations, where inputs affect external actions or critical operations. Building on this insight, we automate the detection of input validation gaps by analyzing existing tests in terms of symbolic constraints derived from the code. For specified input variables, we track the constraints as they propagate toward sink locations, thereby capturing how validation rules influence observable behavior. We then interpret these constraints as equivalence classes represented in the current test suite and use them to identify missing tests. The key idea is simple: if a potential test would produce weaker constraints at a sink location than any existing test, this suggests a path where a validation check may be missing. This approach provides a systematic way to assess and improve the completeness of input validation.

In summary, this paper makes two main contributions: (1) we propose a formal criterion for detecting incompleteness in input validation test suites, and (2) we illustrate its practical relevance through small motivating examples and formal reasoning.

The rest of the paper is organized as follows. Section 2 introduces motivating examples. Section 3 describes the proposed method, including the incompleteness criterion and the test synthesis procedure based on symbolic constraint analysis. Section 4 discusses related work, and Sect. 5 concludes the paper and outlines future work.

2 Motivational Examples

To show the limitations of traditional coverage criteria in assessing input validation completeness, we consider two small programs together with their test suites. In both cases, standard structural criteria can be fully satisfied, but some constraint regions at the sink locations are still not exercised. This means that parts of the effective input domain—namely those that affect observable behavior—remain uncovered.

```
01. void bar(int x, bool b)
{
02.     int res = x;
03.     if(b == true) {
04.          if(x < 1)
05.              res = 1;
06.          if(x > 10)
07.              res = 10;
08.     }
09.     if(res == 10)
10.        report("maximum");
11.   process(res);
12.}
```

Fig. 1. Example 1

```
01.void foo(int d, int n, bool
b){
02.     double res = 1.0;
03.     if(d > 0 || b == true) {
04.          if(n > 0)
05.              res = div(n, d);
06.          else
07.              res = 0.0;
08.     }
09.     send(res);
10.}
```

Fig. 2. Example 2

For the program in the first example (Fig. 1), coverage according to the all-uses and MC/DC criteria can be achieved using the following test suite:

$$\{\ \#1{:}(0,\texttt{True}),\ \ \#2{:}(11,\texttt{True}),\ \ \#3{:}(10,\texttt{False})\ \}.$$

From the data-flow coverage perspective, for the variable `res`, the first test covers the definition–use pair (05, 11), the second covers (07, 09) and (07, 11), and the third covers (02, 09) and (02, 11). The MC/DC criterion is trivially satisfied, as the conditions contain only single predicates, and all branches are covered. From the equivalence class testing perspective, the value of variable `x` falls into the following classes: $x < 1$ (test #1), $x > 10$ (test #2), and $1 \leq x \leq 10$ (test #3). One might assume that the intended range of acceptable values of `res` at line 11 is [1..10]. As a result, these tests trigger a call to the function `process` for only two input values: 1 (test #1) and 10 (tests #2 and #3). All other possible ranges (including $x < 1$ and $x > 10$) remain untested, even though they could be exercised by inputs of the form (x, False) where $x \neq 10$. This naturally raises the question of whether this situation reflects merely missing tests or whether it exposes a defect in the program's validation logic. Note that our method does not attempt to decide program correctness, nor does it explicitly identify sink-specific critical values; rather, it identifies potential input ranges at relevant sink locations that are not covered by the existing tests, thereby guiding the developer's further inspection.

For the program in Fig. 2, the test suite below fully satisfies MC/DC and all-uses:

$$\{\ \#1{:}(0,\texttt{False},0),\ \ \#2{:}\ (1,\texttt{False},1),\ \ \#3{:}(0,\texttt{True},0)\ \}.$$

Test#1 covers the definition–use pair (02, 09) for the variable `res`; test#2 covers (05, 09); and test#3 covers (07, 09). In addition, test#2 covers the computational-use pairs (01, 05) for the variables `d` and `n`. Taken together, these tests satisfy the MC/DC criterion: for the condition at line 03, tests #1 and #2 isolate the effect of the predicate `d` > 0, whereas tests #1 and #3 isolate the effect of the predicate `b`$= =$ `true`. However, none of the tests exercises the case in which the value of `d` passed as an argument to `div` lies in the range `d` ≤ 0. An input such as `d`$= 0$ could—assuming `div` performs an arithmetic division—result in a runtime failure due to division by zero.

3 Method Description

3.1 Test Suite Incompleteness Criterion

Let P be a deterministic program represented as a control-flow graph $G = (N, E)$, where N is the set of statements and $E \subseteq N \times N$ is the set of control-flow edges. Given an input vector *in*, execution of P produces a *concrete trace* $t^c_{in} = <s_0, s_1, \ldots, s_n>$, where $s_i \in N$ and $(s_i, s_{i+1}) \in E$, representing the sequence of executed statements along with concrete variable valuations. Since the program is deterministic, each input *in* induces exactly one feasible trace. An *abstract trace* $t^a_{in} = abstraction(t^c_{in})$ traverses the same control flow path as t^c_{in} and abstracts its execution by replacing concrete values with symbolic expressions over program variables [11] while treating the inputs as non-deterministic assignments. Let T denote the set of abstract traces corresponding to the tests in an

existing test suite, v denote a set of certain (tainted) variables, and let $R(v, s, t)$ denote the symbolic constraints on program variables from v at location $s \in t$ on the trace $t \in T$. Intuitively, these constraints characterize the possible values of variables in v that could be exercised by the sequence of statements traversed by t.

For fixed v and s, we say that the test suite T is ***incomplete*** if there exists a feasible abstract trace t' such that the following condition holds:

$$\forall \, \mathrm{t} \in T, R(v, s, t') \not\models R(v, s, t) \tag{1}$$

This condition indicates that t' explores a path with validation checks not covered by any existing trace in T. Consequently, t' exposes a previously uncovered region of the input space, corresponding to a new equivalence class from the perspective of equivalence class testing. This reasoning builds on the monotonicity property for deterministic programs, according to which the set of feasible input valuations can only decrease along a symbolic path [10]. Formally, for two abstract traces t_1 and t_2 reaching the same program location, $R(v, s, t_1) \vDash R(v, s, t_2) \Rightarrow \mathrm{Inputs}(t_1) \subseteq \mathrm{Inputs}(t_2)$. Applying this property to the above condition, we derive the following implication:

$$\forall t \in T, R(v, s, t') \not\models R(v, s, t) \Rightarrow \forall t \in T, \ \mathrm{Inputs}(t\prime) \nsubseteq \mathrm{Inputs}(t),$$

which formally justifies that t' corresponds to a previously uncovered equivalence class in the input space, and therefore the corresponding test should be added to the suite.

In Example 1, for the variable `res` at line 11, the three test traces yield the constraints `res` $\in \{1, 10\}$. The input `(0,False)` traverses the trace $t' = (01, 02, 03, 09, 11, 12)$, whose abstraction produces constraint $R(\{\texttt{res}\}, 11, t')\equiv \texttt{res} \neq \texttt{10}$. Clearly, this does not imply the constraints observed in tests #1–#3. In Example 2, the input `(0, True, 1)` reveals an unconstrained use of the variable `d`, indicating that `d` lacks prior validation.

The proposed criterion can also be viewed as a logical completeness relation over the symbolic constraints induced by the test suite: if no such t' exists, the test suite is logically complete, i.e., all equivalence classes at the selected location s and for the variable set v are represented by corresponding input values.

We also consider the complementary cases, where the inferred constraints are either *stricter* than those covered by the existing test suite or *partially overlapping* with them.

The first case occurs when, for a fixed variables set v and location s, there exists a feasible trace t' such that the following condition holds:

$$\exists t \in T : \ R(v, s, t') \vDash R(v, s, t) \wedge R(v, s, t') \not\models R(v, s, t') \tag{2}$$

This means that the constraint $R(v, s, t')$ is strictly stronger than the constraint induced by at least one existing test, and thus t' represents an execution in which the validation imposes stricter input conditions than t. Such cases may indicate over-restrictive input validation, where the program unintentionally rejects valid inputs, or inconsistencies between alternative validation paths.

The second case concerns partially overlapping constraints among existing traces.

In this situation, for a fixed set of variables v and location s, there exist traces such that the constraints intersect without fully subsuming each other:

$$\exists t_{\mathrm{i}}, t_{\mathrm{j}} \in T, \ (R(v, s, t_{\mathrm{i}}) \wedge R(v, s, t_{\mathrm{j}}) \neq \bot) \wedge R(v, s, t_{\mathrm{i}}) \not\models R(v, s, t_{\mathrm{j}}) \wedge R(v, s, t_{\mathrm{j}}) \not\models R(v, s, t_{\mathrm{i}}) \tag{3}$$

From the perspective of input validation, such cases may indicate that additional variables should be considered to strengthen the analyzed constraints. However, such overlap may also point to inconsistencies in how validation rules are applied to the same subset of input values across different execution paths. This, in turn, motivates the generation of new tests that explore input combinations both within and outside the overlapping regions. Such tests can help identify parts of the input space where the implemented validation logic may be too restrictive.

3.2 Example

The following example demonstrates how the proposed coverage criterion operates in the case of numeric ranges. Let the initial test suite for Example 3 (Fig. 3) be constructed according to the following set of abstract traces: {$\mathbf{t_1}$: Start, I, A, B, D, S, End_1; $\mathbf{t_2}$: Start, I, A, C, D, S, End_1; $\mathbf{t_3}$: Start, I, F, G, D, E, End_1; $\mathbf{t_4}$: Start, I, F, H, End_2}.

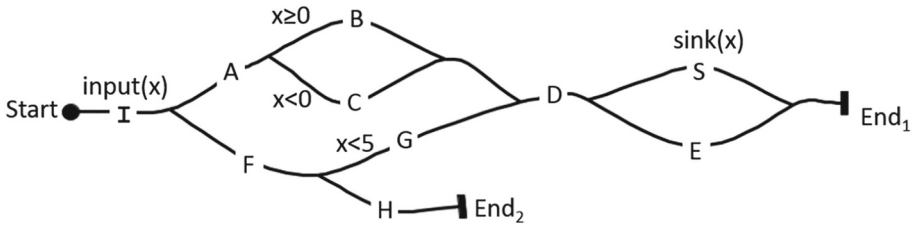

Fig. 3. Example 3

It is worth noting that this set is sufficient to satisfy both the all-uses and branch coverage criteria. The first two traces reach the sink location with the following symbolic constraints on the variable x: $R(x, S, t_1) \equiv x \geq 0$, $R(x, S, t_2) \equiv x < 0$. Taken together, these constraints cover the entire numeric domain. However, there exists an additional feasible trace: $\mathbf{t_5}$: Start, I, F, G, D, S, End_1, which reaches the sink location under the constraint $R(x, S, t_5) \equiv x < 5$. This constraint clearly satisfies condition (1), and according to our criterion, the presence of this trace indicates an incompleteness in the current test suite and therefore it must be incorporated. If, however, the constraint at G were $x = 5$, condition (2) would apply. Similarly, condition (3) could be demonstrated with a constraint like $-5 < x < 5$.

Although the motivating examples use scalar numeric inputs, the proposed method is based on symbolic constraints and is not tied to a specific data type. The notion of a range is treated as an abstract constraint domain, where intersection corresponds to constraint satisfiability. This makes it possible to apply the approach to other structured data types supported by modern SMT solvers. For example, in the case of strings or sequences, constraints may describe length bounds, prefix or substring relations, character-level properties, or other declarative conditions derived from path-specific input validation logic.

Furthermore, the analysis is sink-oriented and relies on function-level symbolic summaries. These summaries can be reused across different calling contexts, which enables compositional reasoning and supports modular analysis.

3.3 Synthesizing New Tests via Symbolic Constraint Analysis

To initiate the search for a potential new test t', we begin with the symbolic constraints derived from the existing test suite T. Each test $t \in T$ that reaches a given program location s defines a constraint $R(v, s, t')$ for a selected set of variables v. The goal is to identify a new test case $t' \notin T$ through a corresponding constraint $R(v, s, t')$ that satisfies one of the following conditions: (1), (2), or (3). Practically, this can be done by:

1. Selecting a representative sink location s and variable set v;
2. Reconstructing the constraints (v,s,t) from traces of existing test suite;
3. Synthesizing candidate constraints based on conditions (1), (2) or (3);
4. In the presence of loops or recursion, the proposed criterion can be applied similarly to existing methods based on bounded unrolling [12]. When extending the current test suite, the same unrolling bound can be used to ensure a fair comparison and consistency with the existing tests.
5. Solving the resulting constraint system to obtain concrete input values for t', typically using an SMT or SAT solver.

If the solver produces a satisfying assignment, it defines a feasible execution trace [11–14] corresponding to a new test case, thereby extending the validation assessment.

One way to synthesize candidate constraints is to use model checking [15]. In this approach, a coverage obligation is encoded as a temporal property over a finite-state model of the program. Using Linear Temporal Logic (LTL), the coverage requirement for a single range r can be written as follows:

$$\varphi_{u,v,\mathrm{r}} = \mathbf{F}(\mathrm{at_u} \wedge (v \in r)).$$

where $\mathrm{at_u}$ denotes the program counter at statement u, and $\mathbf{F}$ is the eventually operator in LTL. While this formulation assumes a fixed target range r, the same mechanism can be extended to support on-the-fly discovery of previously uncovered value ranges. Let `RANGES` denote the set of value ranges already covered by an existing test suite. We use `constraint`(u, v) to denote the set of symbolic constraints over variable v at location u along a given execution path, derived from the program logic and the current path conditions. We write `ranges(c)` for the set of value ranges induced by a constraint `c`. At each iteration, the model checker is queried with:

$$\mathrm{F}(\mathrm{at}_u \wedge ((\exists\, r \in \texttt{ranges}(\texttt{constraint}(u, v)) : r \setminus \texttt{RANGES}(u, v) \neq \varnothing))$$

Intuitively, the query checks whether there exists a feasible execution in which the value of v at location u falls into a range that has not yet been exercised. If the formula is satisfiable, the model checker returns a counterexample execution path. This path can then be translated into a concrete test case. The symbolic constraints associated with this path are used to extract one or more previously uncovered value ranges, which are then merged into `RANGES`. In this way, the set of covered ranges is gradually refined. At each iteration, only ranges that are still uncovered are considered, which helps avoid redundant exploration.

The procedure iterates until the corresponding LTL query produces no counterexample. At this point, the model checker determines that no feasible execution exists in

which v takes a value outside the ranges already represented in RANGES. This means that all distinct value ranges that can reach the use location along feasible program paths have been exercised. Loops do not introduce additional coverage obligations once a fixpoint in the value ranges is reached. After that point, further iterations do not produce new ranges beyond the already established partitions.

This iterative refinement process is conceptually similar to counterexample-guided abstraction refinement (CEGAR) [16]. The set of covered value ranges can be seen as a current abstraction of the input space. Each LTL query then checks whether there exists a counterexample that violates this abstraction. If such a path is found, it is used to refine the abstraction by introducing new value ranges. Unlike standard CEGAR approaches, which typically refine control-flow or predicate abstractions, our method focuses directly on refining value partitions derived from symbolic constraints. As a result, the approach is inherently data-driven.

The procedure terminates due to two key properties. First, refinements are monotonic: each step only extends the accumulated value ranges in RANGES and never removes them. Second, the exploration operates over a finite set of symbolic states modulo constraint equivalence. Therefore, only a finite number of distinct maximal value ranges can be generated at each use location. Once the maximal range at the sink location is reached, further exploration does not produce additional refinements.

Overall, the approach avoids the need to enumerate all possible combinations of value ranges in advance. Instead, it ensures that all distinct value ranges reaching a use location are eventually covered through a systematic, iterative process guided by counterexamples from the model checker.

4 Case Study: Railway Speed Supervision Logic

System Context. Modern railway signaling and train protection systems, such as the European Train Control System, continuously monitor train speed and distance to ensure that a train can safely stop before reaching a target location. These systems compute dynamic braking curves based on the train's braking capability, its current speed, and the remaining distance to the target. The onboard controller predicts future speed reductions using mathematical models of braking dynamics and track characteristics [17]. In many signaling systems, the permitted speed can be approximated using a simple relation derived from braking physics: $v_{safe} = \sqrt{2ad}$, where v_{safe} is the maximum safe speed, a is the braking deceleration, d is the remaining distance to the target. Similar relations appear in railway control systems such as LZB and ETCS when computing permitted speed curves during braking supervision [18]. In this case study, we analyze a simplified supervisory module (Fig. 4) inspired by such speed-monitoring logic.

Supervisory Module Example The module computes a safe speed limit based on current operating parameters. The function `computeSafeSpeed` contains the sink location of interest – the braking-based speed calculation (lines 30, 31). Before performing the calculation, the system validates the inputs using `validateSupervisionInputs` function to ensure that they satisfy operational constraints. The validation logic contains multiple interacting constraints over several input variables, therefore different execution paths impose different constraints on the parameter `currentSpeed` before

reaching this sink. These constraints induce complex regions in the input space, and structural coverage criteria do not guarantee that test cases explore the boundaries of these regions.

```
01. enum Mode { NORMAL, EMERGENCY };
02. bool validateSupervisionInputs(
03.     double currentSpeed,
04.     double targetDistance,
05.     double brakeDeceleration,
06.     Mode mode) {
07.     if (currentSpeed < 0 || currentSpeed > 140)
08.         return false;
09.     if (mode == EMERGENCY && currentSpeed > 30)
10.         return false;
11.     if (targetDistance <= 0)
12.         return false;
13.     if (brakeDeceleration < 0.3)
14.         return false;
15.     return true;
16.}

17. double computeSafeSpeed(
18.     double currentSpeed,
19.     double targetDistance,
20.     double brakeDeceleration,
21.     Mode mode) {
22.     if (!validateSupervisionInputs(
23.             currentSpeed,
24.             targetDistance,
25.             brakeDeceleration,
26.             mode)) {
27.         triggerFailSafeMode();
28.         return 0;
29.     }
30.     return min(currentSpeed,
31.                sqrt(2*brakeDeceleration*targetDistance));
32.}
```

Fig. 4. Supervisory Module Example

Existing Test Suite The test suite shown in Table 1 represents the initial test suite provided as input to our method.

Coverage Achieved The test suite satisfies branch coverage, MC/DC, and all-uses data-flow coverage, with all branches and variable uses exercised. For the compound predicate at line 09, tests T3 and T5 demonstrate the independent influence of `currentSpeed > 30`, while tests T4 and T5 demonstrate the independent influence of `mode == EMERGENCY`. From the perspective of traditional structural testing criteria, the module therefore appears to be adequately tested. However, only test T6 reaches the

Table 1. Existing test suite

test	currentSpeed, m/s	targetDistance, m	brakeDeceleration, m/s^2	mode	validation result
T1	−1	1000	0.1	NORMAL	failed
T2	141	600	0.7	NORMAL	failed
T3	30	0	0.3	EMERGENCY	failed
T4	140	300	0.2	NORMAL	failed
T5	140	200	0.8	EMERGENCY	failed
T6	20	400	1.1	EMERGENCY	successful

speed-computation sink, whereas tests T1–T5 terminate in the validation stage. Consequently, one more range of valid values at the sink location remains uncovered by the test suite. The validation logic defines the following allowable ranges of `currentSpeed` depending on the execution path:

Path 1 — Normal operation:

$$\texttt{mode != EMERGENCY}, 0 \leq \texttt{currentSpeed} \leq 140$$

Path 2 — Emergency supervision

$$\texttt{mode == EMERGENCY}, 0 \leq \texttt{currentSpeed} \leq 30$$

The only test reaching the sink location is:

$$\texttt{T6: mode = EMERGENCY}, \texttt{currentSpeed = 20}$$

The corresponding abstract trace $T6^a$ reaches the sink location under the constraints:

$$\texttt{mode == EMERGENCY} \land 0 \leq \texttt{currentSpeed} \leq 30 \land \texttt{targetDistance} > 0 \land \texttt{brakeDeceleration} \geq 0.3$$

The corresponding LTL query, with the range $\texttt{currentSpeed} \leq 30$ subtracted, produces possible paths reaching the sink location under the given constraints:

$$\texttt{mode == NORMAL} \land 0 \leq \texttt{currentSpeed} \leq 140 \land \texttt{targetDistance} > 0 \land \texttt{brakeDeceleration} \geq 0.3$$

Therefore the following valid input region is not covered:

$$R(\texttt{currentSpeed}, \text{line } 30, \texttt{T7}) = 30 < \texttt{currentSpeed} \leq 140$$

This region corresponds to legitimate operating conditions under the `NORMAL` mode but is never exercised in the computation of the safe speed.

Missing Test Case. Test suite analysis reveals an additional test case (see Table 2) which reaches the sink and exercises the previously uncovered range.

Table 2. Missing test case

test	currentSpeed, m/s	targetDistance, m	brakeDeceleration, m/s^2	mode	validation result
T7	140	10000	1	NORMAL	successful

Summary This case study illustrates how our method identifies uncovered operational ranges in safety-critical control logic, even when the existing test suite satisfies stringent structural coverage criteria such as MC/DC and all-uses dataflow coverage.

5 Related Work and Discussion

Earlier work on test coverage metrics has demonstrated that high structural coverage does not necessarily imply adequate testing of program behaviors, motivating semantic approaches to test adequacy [5, 19]. Reference [20] studies quality attributes of test cases and test suites and identifies coverage as an important concern from practitioners' perspective. These results point to the need for evaluating test adequacy beyond structural metrics, which motivates semantics-aware and specification-independent approaches such as the one proposed in this work.

Apart from verifying known sink-specific values, input validation is often tested indirectly. Practitioners usually rely on structural coverage criteria to ensure that different execution paths are exercised. Our approach follows a related best practice by using equivalence-class reasoning to select representative (boundary) inputs from each class that may lead to different program behaviors [2, 21–23]. In black-box testing, such partitions are typically defined manually. In contrast, techniques such as symbolic execution [13, 14] and coverage-guided fuzzing [24] attempt to explore feasible paths more systematically. However, these approaches do not directly evaluate whether validation is sufficient and provide limited semantic insight into whether executed paths correspond to well- or poorly-validated inputs. Compared to taint-analysis methods [25], which mainly track syntactic propagation through assignments and control dependencies, our approach shifts the focus away from the mere presence of sanitization and toward the logical diversity of input validation.

Work such as [3, 8, 26] employ combinations of validation checks, yet suffer from redundant enumeration and do not evaluate validation completeness. In contrast, our method proceeds in the opposite direction: starting from sink locations, we infer the effective constraints that are actually enforced on the values reaching these points.

Our work is related to, but fundamentally differs from, classical data-flow adequacy criteria [7]. Traditional approaches typically ignore predicate uses along paths from defs to uses, even though guards restrict feasible variable values. As a result, standard data-flow criteria fail to capture how predicates partition the input domain. In contrast, we preserve the predicate-induced constraints on the value of the relevant variable along def-to-use path and, without exhaustive du-path tracking, target equivalence classes observed at sink (i.e., computational use) locations.

Traditional equivalence-class testing partitions the input domain top-down and derives test cases for each partition. Conversely, our approach operates bottom-up, reconstructing the implicit constraints enforced by the existing test suite at selected validation sinks. Rather than focusing on structural coverage, the method directly assesses the logical completeness of input validation. It can be seen as a form of reverse equivalence-class analysis that reveals how the program partitions its input space in practice. Missing tests identified by our criterion often correspond to under-validated or unexpected values at the sink. These cases may reveal real defects, such as insufficient range checks, overlooked corner cases, or latent sanitization errors.

While checking path reachability and comparing symbolic constraints across executions is generally undecidable, practical applicability can still be achieved by restricting the analysis to decidable theories and bounding loops. Under these conditions, SMT solvers or abstract-interpretation techniques can effectively check subsumption and synthesize inputs for previously uncovered regions. Moreover, the analysis can be performed incrementally, guiding the systematic augmentation of the test suite and progressively improving input validation coverage.

6 Conclusions

In this work, we formalize a criterion for identifying incompleteness in existing test suites with respect to input validation by systematically comparing symbolic path constraints across execution traces. The proposed criterion provides a semantics-aware notion of validation coverage, enabling reasoning about gaps in test suites without requiring any external specification, as it operates solely on symbolic abstractions of the program code and the available tests. By operating at the level of symbolic constraints, the approach shifts the focus from syntactic coverage metrics to differences in induced input domains.

The criterion is defined directly in terms of constraint satisfiability and logical implication, which makes it amenable to automated SAT/SMT reasoning. In particular, discrepancies between constraint sets characterize semantically distinct validation behaviors and may also expose logical inconsistencies or unintended over- or under-approximation in the implemented validation logic. This establishes a formal basis for detecting both missing tests and latent defects within the validation component.

Based on this foundation, we present a test synthesis method that constructs new inputs by solving constraint differences derived from symbolic encodings of program paths, providing a mechanism for extending test suites with meaningfully distinct test cases. The method complements traditional structural coverage techniques by focusing on differences in validation behavior rather than simply increasing syntactic coverage. By exposing previously untested validation behaviors, the approach enables a more complete assessment of input validation logic. This is especially relevant for reliability assurance in systems where input validation forms a critical defensive layer.

Future work will focus on improving the efficiency of detecting missing test cases. We also plan to explore integration with automated verification and testing frameworks to support broader use in formally grounded development processes.

References

1. Common Weakness Enumeration (CWE). https://cwe.mitre.org/. Accessed March 2026
2. Homès, B.: Fundamentals of Software Testing, p. Hoboken. John (2024)
3. Liu, H., Tan, H.B.K.: Covering code behavior on input validation in functional testing. Inf. Softw. Technol. **51**(2), 546–553 (2009)
4. Hayes, J.H., Offutt, J.: Input validation analysis and testing. Empir. Softw. Eng. **11**(4), 493–522 (2006). https://doi.org/10.1016/j.jss.2007.05.007
5. Gay, G., Staats, M., Whalen, M., Heimdahl, M.: The risks of coverage-directed test case generation. IEEE Trans. Softw. Eng. **41**, 803–819 (2015). https://doi.org/10.1109/TSE.2015.2421011
6. Chilenski, J., Miller, S.: Applicability of modified condition/decision coverage to software testing. Softw. Eng. J. **7**(5), 193–200 (1994)
7. Rapps S., Weyuker E.: Data flow analysis techniques for test data selection. In: Proceedings of International Conference of Software Engineering, pp. 272–277 (1982)
8. Kolchin, A., Potiyenko, S.: Extending data flow coverage to test constraint refinements. In: ter Beek, M.H., Monahan, R. (eds.) Integrated Formal Methods, IFM 2022. LNCS, vol. 13274. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-07727-2_17
9. Kolchyn, O., Rudenko, D.: Misguarded computations: a pattern of ineffective conditional protection. In: Deshmukh, J., Havelund, K., Pinto, A. (eds.) NASA Formal Methods. NFM 2026. Lecture Notes in Computer Science, vol. 16622. Springer, Cham (2026). https://doi.org/10.1007/978-3-032-28079-4_16
10. Bhatt D., et al.: Requirements-driven model checking and test generation for comprehensive verification. In: NASA Formal Methods Symposium, pp. 576–596 (2022). https://doi.org/10.1007/978-3-031-06773-0_31
11. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
12. Biere, A.: Bounded model checking. In: Handbook of Satisfiability, pp. 739–764. IOS Press (2021)
13. Kurian, E., Briola, D., Braione, P., Denaro, G.: Automatically generating test cases for safety-critical software via symbolic execution. J. Syst. Softw. **199**, 111629 (2023). https://doi.org/10.1016/j.jss.2023.111629
14. Trabish, D., Mattavelli, A., Rinetzky, N., Cadar, C.: Chopped symbolic execution. In: Proceedings of the 40th International Conference on Software Engineering, pp. 350–360 (2018). https://doi.org/10.1145/3180155.3180251
15. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking, vol. 10, pp. 978–983. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8
16. Clarke, E., et al.: Counterexample-guided abstraction refinement. In: Computer Aided Verification. LNCS, vol. 1855. Springer (2000). https://doi.org/10.1007/10722167_15
17. European Union Agency for Railways: Braking curves. https://www.era.europa.eu/sk/node/1831. Accessed March 2026
18. Friman, B.: An algorithm for braking curve calculations in ERTMS train protection systems. Adv. Train Control Syst. **65** (2010). https://doi.org/10.2495/CR060421
19. Bently, W.G., Miller, E.F.: Ct coverage—initial results. Softw. Qual. J. **2**, 29–47 (1993). https://doi.org/10.1007/BF00417425
20. Tran, H.K.V., Ali, N.B., Unterkalmsteiner, M., et al.: Quality attributes of test cases and test suites – importance & challenges from practitioners' perspectives. Softw. Qual J. **33**, 9 (2025). https://doi.org/10.1007/s11219-024-09698-w
21. Huang, W., Peleska, J.: Complete model-based equivalence class testing. Int. J. Softw. Tools Technol. Transfer **18**, 265–283 (2016). https://doi.org/10.1007/s10009-014-0356-8

22. Guo, X., Okamura, H., Dohi, T.: Optimal test case generation for boundary value analysis. Softw. Qual. J. **32**, 543–566 (2024). https://doi.org/10.1007/s11219-023-09659-9
23. van den Bos, P., Tretmans, J.: Coverage-based testing with symbolic transition systems. In: Beyer, D., Keller, C. (eds.) Tests and Proofs. TAP 2019. Lecture Notes in Computer Science, vol. 11823. Springer (2019). https://doi.org/10.1007/978-3-030-31157-5_5
24. Herrera, A., Payer, M., Hosking, A.: DatAFLow: toward a data-flow-guided fuzzer. ACM Trans. Softw. Eng. Methodol. **32**(5), 31 (2023). https://doi.org/10.1145/3587156
25. Alashjee, A.M., Duraibi, S., Song, J.: Dynamic taint analysis tools: a review. Int. J. Comput. Sci. Secur. **13**(6), 231–244 (2019)
26. Liu, H.: Automated recovery, maintenance and testing of input validation. Doctoral dissertation, Nanyang Technological University, 164p. (2009)