MSC 68N30; 68M15 DOI:10.34229/2707-451X.26.2.9

MODEL-BASED TEST CASES GENERATION FOR RANGE-SENSITIVE DATAFLOW COVERAGE

Oleksandr Kolchyn, Daria Rudenko *

ORCIDs: 0000-0001-7809-536X, 0009-0008-7499-5895

V.M. Glushkov Institute of Cybernetics of the NAS of Ukraine, Kyiv
* *Correspondence: kolchin_av@yahoo.com*

УДК 004.415.53

Модельно-орієнтована генерація тестів для покриття потоку даних з урахуванням діапазонів значень

О.В. Колчин, Д.Б. Руденко *

Інститут кібернетики імені В.М. Глушкова НАН України, Київ
* *Листування: kolchin_av@yahoo.com*

Вступ. Критерії покриття відіграють ключову роль у програмному тестуванні, забезпечуючи об'єктивні показники адекватності тестових наборів та слугуючи основою для систематичної генерації тестів. Структурні критерії, зокрема покриття операторів, гілок і потоків даних, дозволяють перевіряти керуючі та інформаційні залежності в програмі. Водночас класичні методи, орієнтовані на значення, такі як еквівалентне розбиття та аналіз граничних значень, зазвичай застосовуються лише до зовнішніх вхідних даних і не враховують варіативність значень усередині програми.

У багатьох сучасних програмних системах поведінка визначається не лише структурою керування або фактом досягнення використання змінних, а й конкретними діапазонами значень, що поширюються вздовж шляхів виконання. Класичні критерії покриття потоків даних, зокрема критерій all-uses, вважають пару «визначення–використання» покритою після будь-якого її виконання, не розрізняючи значення, з якими змінна досягає точки використання. Унаслідок цього значущі, але вузькі або специфічні діапазони значень можуть залишатися неперевіреними навіть за досягнення повного структурного покриття.

Ця проблема є особливо актуальною для програм, чутливих до вхідних даних, де коректність залежить від порогових значень, числових обмежень або прихованих припущень щодо допустимих діапазонів. Сучасні підходи до автоматизованої генерації тестів, зокрема символічне виконання та модельна перевірка, частково враховують обмеження на значення, проте їхні цілі покриття здебільшого залишаються структурно орієнтованими. Це створює розрив між аналізом потоків даних і систематичним тестуванням значень, що зумовлює необхідність розроблення більш семантично чутливих критеріїв покриття.

Мета роботи – розробка та обґрунтування критерію покриття потоків даних, який ураховує семантично відмінні діапазони значень змінних у точках використання, а також створення модельно-орієнтованих методів генерації тестових випадків для досягнення такого покриття.

Результати. У роботі запропоновано розширення критерію покриття потоку даних шляхом урахування діапазонів значень у точках використання, а також описано два підходи до його досягнення: на основі символічного моделювання; із застосуванням модельної перевірки з використанням LTL-формул. На прикладах показано як запропонований підхід дає змогу виявляти дефекти, які залишаються непоміченими за використанням традиційних критеріїв.

Висновки. Запропонований підхід поєднує структурний аналіз потоків даних із явним урахуванням варіативності значень, забезпечуючи більш семантично насичене тестове покриття. Критерій доповнює, а не замінює існуючі підходи, та є перспективним для застосування у тестуванні коректності валідації вхідних даних, у виявленні потенційних вразливостей, а також у тестуванні кібер-фізичних систем. Хоча досягнення такого покриття може бути обчислювально затратним, його застосування є доцільним для критичних точок використання, пов'язаних із обробленням вхідних даних та перевіркою обмежень.

Ключові слова: генерація тестових сценаріїв, аналіз потоку даних, критерії тестового покриття.

Abstract

Coverage criteria play a central role in software testing by providing objective measures of test suite adequacy and serving as a foundation for systematic test generation. Structural criteria – such as statement, branch, and dataflow coverage – enable verification of control and information dependencies within a program. At the same time, classical value-based techniques, including equivalence partitioning and boundary value analysis, are typically applied only to external inputs and do not account for the variability of values propagated within the program's internal state.

In many contemporary software systems, behavior is determined not only by control-flow structure or by the mere fact that variable uses are reached, but also by specific ranges of values propagated along execution paths. Traditional dataflow coverage criteria, in particular the *all-uses* criterion, consider a definition–use pair to be covered if it has been executed at least once, without distinguishing the values with which the variable reaches the use site. As a result, semantically significant yet narrow or domain-specific value ranges may remain untested, even when full structural coverage has been achieved.

This limitation is especially critical for input-sensitive programs whose correctness depends on threshold conditions, numeric constraints, or implicit assumptions about admissible value ranges. Modern approaches to automated test generation, such as symbolic execution and model checking, partially account for value constraints; however, their coverage objectives typically remain structurally oriented. This creates a gap between dataflow analysis and systematic value-based testing, motivating the development of coverage criteria that are more sensitive to semantic distinctions.

Objective. The goal of this work is to develop and formally justify a dataflow coverage criterion that accounts for semantically distinct value ranges of variables at use sites, and to design model-based methods for test case generation aimed at achieving such coverage.

Results. We propose an extension of classical dataflow coverage by incorporating value-range distinctions at variable use points. Two complementary approaches to achieving this coverage are described: one based on symbolic modeling, and another based on model checking with Linear Temporal Logic specifications. Illustrative examples demonstrate that the proposed method reveals defects that remain undetected under traditional structural coverage criteria.

Conclusions. The proposed approach integrates structural dataflow analysis with an explicit treatment of value variability, thereby providing semantically enriched test coverage. The criterion is intended to complement, rather than replace, existing approaches and appears particularly promising for testing input validation logic, detecting potential vulnerabilities, and also verifying cyber-physical systems. Although achieving such coverage may incur significant computational cost, it is justified for critical use points associated with external inputs and constraint checking.

Keywords: test cases generation, dataflow analysis, coverage criteria.

Introduction. Coverage criteria play a crucial role in software testing, providing objective measures of test adequacy and guiding systematic test generation [1]. Structural criteria such as statement, branch, and dataflow coverage ensure that control-flow constructs and definition-use associations are exercised, while black-box techniques such as equivalence partitioning and boundary value analysis emphasize the importance of testing representative regions of the input domain. Together, these approaches have proven highly effective in practice [2]. However, they operate at fundamentally different levels of abstraction and leave a semantic gap between internal data propagation and value-oriented test objectives: structural criteria reason about program structure while abstracting away from concrete values, whereas value-based techniques are typically applied only at the level of external inputs.

This separation becomes increasingly problematic in programs where observable behavior varies significantly depending on propagated values. A single input-related definition may reach a computation use under numerous semantically distinct value contexts, since it traverses different constraint checks and validation branches, while formally preserving the same def–use relationship. Traditional dataflow criteria, including all-uses coverage [3, 4], treat these executions as equivalent once the def–use pair has been exercised. Consequently, a test suite may satisfy conventional coverage criterion while exercising a computation use only for a narrow subset of feasible values, leaving value-dependent behaviors untested. This limitation

is particularly relevant for faults whose manifestation depends on specific value ranges rather than on control-flow structure alone. Typical examples include threshold violations, arithmetic corner cases, or value-triggered changes in program modes. Such faults may remain undetected even under thorough structural testing [5] if the exercised executions do not traverse the critical regions of the value space. This risk is especially pronounced in input-sensitive numerically intensive systems, where subtle variations in internal values can lead to qualitatively different behaviors or violations of implicit assumptions.Recent advances in automated test generation, especially those based on symbolic and concolic execution [6–8], partially mitigate this issue by reasoning about value constraints along execution paths. These techniques systematically explore path conditions and can generate tests that satisfy specific branch predicates. Nevertheless, their coverage objectives are typically defined in terms of paths, branches, or other structure-oriented features, rather than in terms of value diversity at specific use locations. As a result, they do not explicitly require that all distinct value ranges flowing to a computation use be exercised even along the same feasible path. Similarly, static analyses such as value range analysis can infer possible value sets at program points, but these results are primarily used for verification or bug detection [9], not as explicit coverage obligations guiding test generation.

This paper is motivated by the need to bridge the gap between structural dataflow coverage and value-oriented testing. We introduce Range-Sensitive Data-Flow Coverage (RS-DFC), a coverage criterion that refines classical data flow coverage by incorporating explicit requirements over value ranges. Instead of considering a def–use pair as covered once it is exercised by any execution, RS-DFC treats distinct ranges of values observed at the use location as first-class coverage objectives, rather than incidental by-products of path exploration. A test suite satisfies RS-DFC only if each required value range associated with a def–use pair is exercised by at least one execution.

By integrating value sensitivity directly into the definition of dataflow coverage, RS-DFC provides a more expressive notion of test adequacy while remaining compatible with established coverage principles and complements existing structural criteria, offering a principled foundation for systematically targeting value-dependent behaviors that are beyond the reach of traditional dataflow testing.

Although this work is primarily concerned with software-level coverage, the proposed approach is also naturally applicable to cyber-physical systems, where internal variables often represent physical quantities and distinct value ranges correspond to qualitatively different system behaviors.

To operationalize RS-DFC in practice, we present a model-based test generation approach that incrementally achieves RS-DFC through symbolic exploration of the program state space. Value ranges are obtained from two complementary sources. First, *statically* determined partitions can be derived prior to test generation, based on static program analysis, specifications, type constraints, assertions, or domain knowledge. These partitions guide test generation toward covering all required categories. Second, on-the-fly objective generation *dynamically* identifies previously unobserved value ranges during symbolic exploration and incorporates them as new coverage objectives. Together, these mechanisms enable adaptive and systematic exploration of value-sensitive behaviors, ensuring thorough coverage of both anticipated and emergent value ranges. The main contributions of this paper are as follows:

1. We introduce a novel coverage criterion that refines classical dataflow coverage by treating semantically distinct value ranges at computation uses as explicit coverage obligations;

2. We present a model-based test generation algorithm that incrementally achieves the coverage by combining statically defined value partitions with on-the-fly discovery of path-induced ranges during symbolic exploration.

1. Motivational examples

Example 1. Consider a simple temperature-monitoring program shown in fig. 1. The variable `v`, obtained from the external function `read_temperature`, is subsequently used at several program points.

The calls to `alert_temperature` represent threshold-dependent reactions, while `proceed(v)` corresponds to an external computation use of the measured value.

This program induces several def–use pairs for the variable `v`, corresponding to the threshold-dependent uses at lines 2, 4, and 6, as well as the external computation use at line 8. According to the classical all-uses dataflow coverage criterion, a test suite is required to exercise each of these def–use pairs at least once. For instance, the following two test cases are sufficient to satisfy both branch coverage and the all-uses criterion:

- Test 1.1: `v` = 0, which exercises the execution path where all threshold predicates evaluate to *false* and the external call `proceed(v)` is executed.

- Test 1.2: `v` = 61, which causes all threshold predicates to evaluate to *true* and executes the external call.

Together, these test cases exercise all branches and all def–use pairs of the program. Moreover, since each decision consists of a single condition and both its true and false outcomes are exercised, the test suite also satisfies Modified Condition/Decision Coverage [10] (MC/DC) criterion. Nevertheless, the test suite does not distinguish between intermediate temperature ranges (0,30] and (30,60], even though these ranges correspond to semantically different reactions of the system.

```
1    int v = read_temperature(); // definition
2    if (v > 0)                  // p-use1: temperature above 0
3        alert_temperature(1);
4    if (v > 30)                 // p-use2: temperature above 30
5        alert_temperature(2);
6    if (v > 60)                 // p-use3: temperature above 60
7        alert_temperature(3);
8    proceed(v);                 // c-use: external procedure call
```

FIG. 1. Example of a temperature-monitoring program

Example 2. Consider a program that processes the same logical input obtained from two different sources, such as an external HTTP request and an internal configuration file (fig. 2). Although both sources are intended to provide values within the same valid range (`amount` ∈ [0,100]), explicit validation is applied only to the external input, while the internal source is implicitly trusted. As a result, values outside the intended range may reach the same use location along one def–use path but not the other.

```
01   int amount;
02   if (source == HTTP) {
03      amount = parseHttpParameter();     // def1
04      if (amount < 0 || amount > 100) {
05          report_error();
06          return;
07      }
08   } else {
09      amount = readConfigValue();        // def2
10      /* assumed to be validated */
11   }
12   process(amount);
```

FIG. 2. Example of asymmetry in validation of external inputs

The following test suite is sufficient to satisfy branch coverage, MC/DC, and classical all-uses data flow criterion:

- Test 2.1: `source` = `HTTP`, `amount` = 50, exercising the def–use pair (03,12) under a valid input,

- Test 2.2: `source` ≠ `HTTP`, `amount` = 50 , exercising the def–use pair (09,12),
- Test 2.3: `source` = `HTTP`, `amount` = -1, exercising the predicate `amount` < 0 and the error case,
- Test 2.4: `source` = `HTTP`, `amount` = 101, exercising the predicate `amount` > 100 and the error.

This test suite satisfies all required structural control- and data-flow coverage obligations: both definitions of `amount` reach the use at line 12, and all relevant branch outcomes are exercised. However, the difference in the value ranges reaching the call to `process(amount)` remains unobserved. In particular, no test exercises the execution in which the internally sourced definition (def2) reaches the use with values outside the intended range. Consequently, the asymmetry in input validation remains undetected, despite full satisfaction of traditional coverage criteria.

Example 3. Consider a simplified adaptive cruise control component of an autonomous vehicle, responsible for computing longitudinal acceleration in order to maintain a safe distance to a leading vehicle (fig. 3). The controller operates on physical signals obtained from heterogeneous sensors, whose measurements differ in precision and uncertainty, and applies a feedback control law to determine the actuation command. Here `d` denote the measured distance to the lead vehicle, `d_safe` is a desired safety distance, and `k_d` is a certain coefficient. Distance measurements are obtained either from a radar sensor or from a camera-based perception pipeline:

```
01   float d;           // distance to lead vehicle (m)
03   if (RADAR_AVAILABLE)
04      d = radarDistance();         // high precision
05   else
06      d = cameraDistance();        // noisy estimate
07   float acc = k_d * (d - d_safe); // control law
08   if (emergency)
09      applyEmergencyBrake();       // emergency condition
10   else
11      applyAcceleration(acc);      // acceleration
```

FIG. 3. Example of a cyber-physical system

Although the two sensor modalities share the same control-flow structure, the value ranges induced by the corresponding definitions of `d` differ substantially. Radar-based measurements typically yield relatively narrow and well-calibrated ranges, whereas camera-based source is more susceptible to environmental factors. While their ranges overlap, each sensor also admits values that lie outside the typical range of the other.

The following test suite is sufficient to satisfy branch coverage, MC/DC, and the all-uses:

- Test 3.1: `RADAR_AVAILABLE`, d ∈ D(`radarDistance`)∩D(`cameraDistance`), ¬`emergency`, exercising the def–use pair (04,07) for variable `d` and def-use pair (07,11) for variable `acc`;
- Test 3.2: ¬`RADAR_AVAILABLE`, d ∈ D(`radarDistance`)∩D(`cameraDistance`), ¬`emergency`, exercising the def–use pair (06,07) for variable `d` and def-use pair(07,11) for variable `acc`;
- Test 3.3: `RADAR_AVAILABLE`, d ∈ D(`radarDistance`) \ D(`cameraDistance`), `emergency`, exercising the def–use pair (04,07) for variable `d` and call of `applyEmergencyBrake()` function.

From the perspective of the all-uses criterion, both definitions of `d` reach the same computation use and the same actuation site, and thus a small number of tests suffices to satisfy the coverage criterion. However, such coverage does not ensure that executions exercise value ranges unique to each sensing modality, nor that values exceeding the “competing” range of the alternative definition are explored. In particular, covering the def–use pair via radar-based measurements does not imply that camera-specific value ranges have been exercised.

This example highlights the relevance of range-sensitive dataflow coverage for cyber-physical systems [11, 12] testing, where correctness depends not only on reaching a computation or actuation point, but also on the physical context encoded by the values flowing through the system.

2. Preliminaries

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

Definition 2.1 (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 $\pi = \langle \ell_0,\ell_1,\ldots \rangle$, 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 Def(x) and Use(x) denote its definition and uses nodes, with $PUse(x) \subseteq Use(x)$ for uses in guard predicates, and $CUse(x) \subseteq Use(x)$ 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 2.2 (Reaching Definition). Let $d \in Def(x)$ and $u \in Use(x)$. The definition d of variable x is said to reach the use u, iff: there exists a path $\pi = \langle d,\ldots,u \rangle$ in G, and no node on π redefines x (π is a "def-clear" path with respect to x [13]).

A definition–use association for a variable v is a pair $(d,)$ 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).

Definition 2.3 (Dominating Guard). A node n dominates a node m, written as $n \ll m$, iff every path from ℓ_0 to m passes through n [14].

If $g \ll c$, then the outcome of the guard at g determines whether execution can reach the computational use at c. Consequently, the predicate associated with g induces constraints on the values of variables at c.

3. Range-sensitive dataflow coverage

Let us consider a program P with a set of variables V and let $DU(P)$ denote the set of all def–use pairs in P.

Traditional all-uses coverage requires that for each $(d,u,v) \in DU(P)$, 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 [13] 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 [15]. The range-sensitive dataflow coverage criterion strengthens all-uses coverage by requiring that all meaningful value partitions at each use location are exercised.

Definition 3.1 (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 $R_{v,u} = \{r_1,r_2,\ldots,r_k\}$ 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 5. 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 {true} and {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 $r \in R_{v,u}$ if, during execution with input t, the value of v at u lies within r.

Definition 3.2 (Range-Sensitive Dataflow Coverage, RS-DFC). A test suite $T=\{t_1,t_2,\ldots,t_n\}$ 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.

Remark (Monotonicity and Feasibility). Let $C(T) \subseteq \bigcup_{(d,u,v)\in DU(P)} R_{v,u}$ denote the set of ranges covered by a test suite T. By construction, RS-DFC is *monotone*: for any two test suites T,T′ with $T' \supseteq T$, we have

$C(T') \supseteq C(T)$.

Moreover, RS-DFC admits a well-defined *feasibility* interpretation with respect to a finite set of explicitly specified ranges $R_{v,u}$: the coverage of each range can be determined by executing the tests in T and checking whether the value of v at u lies in the corresponding range. Unreachable or unrealizable ranges in $R_{v,u}$ are considered not covered, analogously to infeasible def–use pairs in classical dataflow coverage, and do not invalidate the definition of the criterion.

Scope and applicability. It is important to clarify the scope of def–use pairs for which the proposed criterion provides additional benefit beyond existing dataflow coverage criteria. First, observe that not all definitions induce meaningful value variability at their corresponding use locations. In particular, if a definition assigns a constant value to a variable, then the set of values that may reach any subsequent use is a singleton, consequently, coverage criteria based on distinguishing value ranges provide no additional testing benefit for such definitions. For this reason, the proposed criterion is primarily concerned with input-dependent definitions, i.e., definitions whose assigned values are not statically fixed and whose concrete values depend on external inputs, sensor readings, environment interactions, computations that propagate previously unconstrained values, or other non-deterministic sources. These definitions may induce a wide (possibly unbounded) domain of values that can reach subsequent uses, making them suitable candidates for range-based partitioning. By the same reasoning, not all uses admit meaningful value variability: if a dominating guard $g \ll u$ restricts the value of a variable to a singleton before the use at u, then the use is considered to admit only a single feasible value and is therefore can be excluded from range-sensitive partitioning.

Furthermore, among all possible def–use pairs, it is reasonable to focus specifically on pairs of the form Def–CUse. Computation uses represent sink locations in the dataflow of a variable, where its value is consumed by an operation that may influence observable behavior, system state, or interactions with the environment. Such locations are often more fault-prone than predicate uses, as they may involve arithmetic operations, external calls, resource manipulation, or safety-critical actions. As a result, insufficient testing of these uses may lead to subtle faults that remain undetected under traditional structural coverage criteria. In contrast, predicate uses (PUse) typically do not require range sensitivity, since they primarily affect control-flow decisions and are already well addressed by existing criteria such as branch coverage or MC/DC.

Note also that RS-DFC does not require covering all def–use paths or all loop iterations. Coverage obligations are defined over def–use pairs and value ranges, not over individual executions.

Discussion. The partitioning yields explicit objectives that are more informative than plain def–use coverage: instead of mere structural coverage, it also requires exercising each relevant value region. Intermediate predicates between the definition location and the destination use location are therefore no longer ignored but actively contribute to the partitioning of the input domain that must be tested.

RS-DFC remains a well-defined and practically measurable coverage criterion: monotonicity ensures that adding tests never decreases coverage, and explicitly considering feasibility clarifies how coverage should be interpreted for all ranges in $R_{v,u}$, including those that are unreachable.

It should also be noted that the requirement to cover all value-range classes at every use location can be prohibitively expensive in terms of test generation and execution. Consequently, the use of RS-DFC is justified mainly in safety-critical applications, input validation routines, and other scenarios where inadequate coverage of value ranges poses a significant risk of failure or unsafe behavior [2, 5, 16].

Regarding Example 1, the two test cases listed are insufficient to satisfy the RS-DFC criterion, as they do not distinguish between intermediate temperature ranges. In particular, additional test inputs are required to cover the intervals (0,30] and (30,60], thereby ensuring that all meaningful value partitions influencing the program's behavior are exercised:

- Test 1.3: `v` = 30, which exercises the execution path with interval (0,30];
- Test 1.4: `v` = 60, which exercises the execution path with interval (30,60].

Regarding Example 2, RS-DFC, by contrast to the listed test cases, distinguishes between the value ranges propagated along each path and requires tests that exercise both the expected and anomalous ranges:

- Test 2.5: `source` ≠ `HTTP`, ¬(0 ≤ `amount` ≤ 100), exercising the def–use pair (09,12).

Regarding Example 3, RS-DFC requires additional tests to exercise value ranges that are specific to each sensing modality and lie outside the overlapping region of their domains:

- Test 3.3: `RADAR_AVAILABLE`, d ∈ D(`radarDistance`) \ D(`cameraDistance`), ¬`emergency`, exercising values outside the camera-induced range for the def–use pair (04,07) for variable `d` and the corresponding def–use pair (07,11) for variable `acc`;
- Test 3.4: ¬`RADAR_AVAILABLE`, d ∈ D(`cameraDistance`) \ D(`radarDistance`), ¬`emergency`, exercising values outside the radar-induced range for the def–use pair (06,07) for variable `d` and the corresponding def–use pair (07,11) for variable `acc`.

In summary, the criterion targets def–use pairs where (i) the definition introduces genuine value variability and (ii) the use corresponds to a computation location whose behavior may be sensitive to different value ranges. By restricting attention to such pairs, the criterion avoids trivial cases while concentrating testing effort on locations that are most likely to benefit from a finer-grained exploration of the input domain.

4. Test cases generation

To enforce Range-Sensitive Data-Flow Coverage, we adopt a model-based test generation approach that systematically derives test inputs covering all required value ranges for relevant def–use pairs. A key challenge in dataflow-based testing – and one that becomes more pronounced for value-sensitive criteria – is the treatment of feasibility.

Classical dataflow coverage criteria are defined over the syntactic structure of the control-flow graph and implicitly assume that def–use pairs are feasible. In practice, however, many def–use pairs cannot be exercised by any execution due to contradictory path conditions or mutually exclusive guards. As a result, coverage requirements may include objectives that are in fact unattainable, complicating both coverage measurement and automated test generation. Determining feasibility of def–use pairs is itself a non-trivial problem, often undecidable in the general case, and therefore commonly addressed through conservative approximations, heuristic pruning, or manual intervention. Existing approaches either ignore infeasible pairs altogether or rely on incomplete static analyses to filter them out. This issue becomes even more acute when the coverage requirement is strengthened with additional distinctions, such as value range partitions. Enumerating all def–use pairs together with their associated value ranges a priori risks generating a large number of infeasible coverage obligations, undermining scalability and obscuring the distinction between unattainable and merely uncovered objectives.

Rather than precomputing all def–use–range combinations and proving their feasibility, our method incrementally discovers feasible value ranges as symbolic constraints are accumulated along execution

paths. Whenever a previously unobserved value range at a computation use is encountered, it is recorded as a covered objective, while unexplored value contexts induce new test goals that guide further exploration. Coverage obligations are therefore generated and discharged only when the corresponding symbolic states are shown to be feasible by the constraint solver.

This dynamic treatment of coverage naturally aligns with symbolic execution [17] and model checking [18] techniques, where infeasible paths are pruned as soon as their constraints become unsatisfiable. As a result, infeasible def–use–range combinations are never materialized as coverage goals, eliminating the need for a separate feasibility analysis phase. At the same time, the algorithm adaptively identifies and covers value ranges that may not have been anticipated statically, complementing pre-defined partitions when available. This combination enables RS-DFC to be achieved without requiring an upfront enumeration of all value partitions, while still guaranteeing that all feasible and behaviorally distinct value ranges reaching computation uses are systematically exercised.

4.1. Symbolic state-space exploration approach

The proposed algorithm can be implemented using symbolic state-space exploration, performing a DFS/BFS over program states and maintaining a visited-state cache to avoid redundant exploration. While the case of statically defined ranges is straightforward, this section focuses on the algorithm for on-the-fly generation of test objectives, where value range partitions are discovered dynamically during program exploration. In this approach, the set of ranges $R_{v,u}$ is derived from path-sensitive constraints along feasible def–use paths, capturing behaviorally distinct value contexts that can reach each use.

For simplicity of presentation, the algorithm is illustrated on a single def–use pair, though it generalizes naturally to the full set of def–use pairs in a program. The program behavior is modeled as a finite-state abstraction modulo symbolic restrictions, assuming that the number of distinct symbolic states considered by the algorithm is finite. The algorithm systematically explores the program's symbolic state space to identify inputs that cover all value range partitions for a definition–use association `(d,u,v)`. It maintains a set of covered ranges and dynamically generates new objectives whenever previously untested ranges are encountered.

Each symbolic state encodes both the program location and the current constraints on variable values, enabling the algorithm to track which value ranges have already been covered. By pruning symbolic states whose ranges have been previously exercised, the algorithm avoids redundant exploration while preserving RS-DFC coverage obligations. A range is considered subsumed if its values are included in a previously covered range at the same program location.

Let $R_{v,u}=\{R_1,\ldots,R_m\}$ denote the value range partition of variable v at location u. Let `PC(p)` denote the path condition associated with a symbolic path `p`. A symbolic path `p` is said to cover range $R_k \in R_{v,u}$ if there exists a state $s \in p$ such that:

$s.loc = u \wedge \exists r \in ranges(s.v.constraint) : r \cap R_k \neq \emptyset$.

Lemma (Soundness of symbolic range coverage). If a symbolic path `p` covers range $r \in R_{v,u}$, then there exists at least one concrete test input *t* instantiating `p` such that *t* covers *r* in the sense of Definition 2.2.

Proof. Since `PC(p)` is satisfiable, there exists $t \vDash$ `PC(p)`. By the semantics of symbolic execution, the value of *v* at *u* under *t* satisfies the symbolic constraint associated with state at the location *u*, and therefore lies in a range intersecting *r*.

In the algorithm description (fig. 4), we use the following auxiliary notations:

`s.loc` – the program location associated with state `s`;

`s.v.def` – the definition location of variable `v` in state `s`;

`s.v.constraint` – the path constraint imposed on variable `v` in state `s`; the constraint may correspond to one or more value ranges;

`ranges(q)` – denotes the set of disjoint value ranges induced by the path constraint `q`;

`restrict(s)` denotes a symbolic constraint characterizing the entire state `s`;

RANGES – the set of value ranges encountered so far;

PATHS – the set of symbolic states reaching previously uncovered value ranges.

We define ranges(s.v.constraint) as the set of maximal value ranges induced by this constraint. Specifically, ranges(s.v.constraint) is obtained by decomposing the set of values satisfying s.v.constraint into a finite collection of disjoint ranges, where each range is a maximal contiguous subset of the domain of v (for ordered domains), or a maximal subset for unordered or discrete domains.

```
01 procedure du_range_search(Initial, Final, d, u, v)
02   PATHS := ∅;
03   RANGES := ∅;
04   WAIT := {Initial};
05   VISITED := ∅;
06   while WAIT≠∅ do
07     select s from WAIT;
08     if (∃s_i: s_i∈VISITED ∧ restrict(s) ⊆ restrict(s_i)∧
09         (s.v.def = s_i.v.def ∨ s.v.def ≠ d)) then
10       continue;
11     add s to VISITED;
12     if s.v.def = d ∧ s.loc = u then
13        for each r in ranges(s.v.constraint) do
14           if R∈RANGES ⇒ ¬(r ⊆ R) then do
15              add s[v ∈ r] to PATHS;
16              update RANGES with r;
17              prolong_path(s[v ∈ r], Final);
18           od
19        od
20     else
21       for all successor states s' outgoing from s
22         add s’ to WAIT;
23   od
24   return {PATHS, RANGES};
```

FIG. 4. Procedure du_range_search

The algorithm is implemented by the procedure du_range_search, which takes as input the initial and final locations together with a def–use pair (d, u, v). It initializes the result sets PATHS and RANGES, and two auxiliary storages: WAIT for unexplored states and VISITED for already processed states. The procedure unfolds paths from the initial state by repeatedly selecting states from WAIT. Whenever a state s is subsumed by a previously visited state s_i (lines 08–10), exploration is terminated to avoid redundant exploration. This check combines both the comparison of symbolic restrictions on variables and the definition location of variable v. The subsumption criterion is analogous to state subsumption in abstract interpretation, where abstract states are ordered by inclusion and exploration terminates once a fixpoint is reached. However, in our case subsumption is driven by coverage obligations rather than fixpoint computation.

When the use location u is reached with a definition d (lines 12–19), the algorithm checks whether the value of v falls into a range not yet covered. In that case, the state is recorded in PATHS, the new range is absorbed by RANGES, and the auxiliary procedure prolong_path is invoked. This procedure ensures

that, once a new value range has been identified at a use-location, the generated path is systematically extended toward the final location of interest while preserving the constraints that characterize the discovered range. In safety-critical testing, boundary coverage is considered essential, since boundary points are particularly error-prone [1, 19]: many program faults manifest exactly at threshold conditions (e.g., $t > 30$ versus $t \geq 30$). The rationale behind `prolong_path` is therefore to guarantee that boundary values of each range are exercised by concrete test cases. By prolonging the symbolic path, the algorithm derives additional test inputs corresponding to the extreme values at the edges of each partition, ensuring that both interior and boundary behaviors of the program are tested [20].

Otherwise, the algorithm continues exploration by adding successor states to `WAIT` (lines 21–23). The procedure terminates when all states are processed, returning the set of paths covering distinct ranges along with the set of discovered ranges. Upon termination, the set `RANGES` represent exactly those ranges that are actually reachable by some execution of P under the symbolic semantics, and these ranges are considered *covered*. Any ranges in $R_{v,u}$ not present in `RANGES` are either unreachable or unrealizable, and are therefore treated as *not covered*.

The following main properties of the algorithm of Figure 4 summarize the description:

Theorem. Let `du_range_search(Initial, Final, d, u, v)` be executed on a program model. Then the following properties hold:

1. *Termination*. If the underlying state space is finite modulo symbolic restrictions, then the algorithm terminates. Its asymptotic time depends on |Q| and |S|, where Q is the number of transitions in the model and S is the number of symbolic states.

2. *Soundness* (Correctness). Every range $r \in$ `RANGES` returned by the algorithm corresponds to an actual execution path from `d` to `u` such that the value of `v` at u belongs to `r`. In other words, no spurious ranges are included.

3. *Completeness* (Range Coverage). Upon termination, if there exists a reachable path π in the model such that it covers a definition-use association `(d,u,v)` and the value of `v` at `u` lies within some range `r`, then there is range $r' \in$ `RANGES` in the result such that $r \subseteq r'$.

Sketch of proof.

Termination. The algorithm terminates when the set `WAIT` is empty. Its size depends on the finite number of transitions and states. Lines 08–10 and 21–23 ensure that no state is unfolded infinitely. For a fixed def–use pair, repeated traversal of a loop cannot introduce new uncovered value ranges once the maximal range induced by the loop has been covered.

Soundness. Follows from the construction: a new range is added (line 16) only when a state `s` reaching `u` with `d` is observed, so each $r \in$ `RANGES` is realizable.

Completeness. The proof proceeds by induction on the number of completed iterations of the main loop (lines 06–23) to show that for every range, either it is already included in `RANGES` or there exists a state in `WAIT` from which this range can be reached.

Base case. By assumption, the desired range is reachable. Therefore, it is reachable from the `Initial` state, which is included in `WAIT` at line 04.

Inductive step. Assume that after $k > 1$ iterations, every range $r \in R_{v,u}$ is either already included in `RANGES` or there exists a state in `WAIT` from which can be reached a state covering $r' \supseteq r$. In iteration k+1, a state `s` is selected from `WAIT`:

Case 1. The unfolding of state `s` is terminated at line 10. This indicates that `s` is already subsumed by a previously visited state s_i (either because s_i`.v.def` = `s.v.def` or `s.v.def` $\neq$ `d` and `restrict(s)` $\subseteq$ `restrict(`s_i`)`). In this case, the range associated with `s` has already been added to `RANGES` at line 16 during the iteration when s_i was processed. The pruning condition is sound due to the monotonicity of path

constraints. In symbolic execution, path conditions are accumulated conjunctively along execution paths, and therefore additional constraints can only refine the set of feasible values of a variable at a given program point, but cannot extend it [21, 17]. As a result, if a value range observed at a symbolic state is included in a previously covered range for the same def–use pair, further exploration of that state cannot yield new uncovered value ranges.

Case 2. The range is not yet covered and is not reachable from `s`. Then, by the inductive assumption, there still exists another state in `WAIT` from which the range is reachable (and thus it will be eventually selected at line 07).

Case 3. State `s` reaches location `u` with definition `d`, covering the required range. Then, according to lines 12–16, a new symbolic range is added to `RANGES`.

Case 4. State `s` does not cover the range, but the range is still reachable from `s`. The algorithm unfolds all outgoing transitions of `s` (lines 21–23). Any path that continues toward `u` is added via successor states to `WAIT`, preserving the inductive invariant.

Since the set of distinct symbolic restriction profiles is assumed to be finite, all reachable ranges are eventually explored and added to `RANGES`. □

4.2. Model checking approach

An alternative way to realize the coverage criterion is through model checking [18]. The idea is to encode each coverage obligation into 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 expressed as:

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

Here, at_u denotes program counter at statement u, and def(v) denotes definition location of variable v.
A test suite $T=\{\pi_1,\ldots,\pi_n\}$ satisfies RS-DFC iff:

$$\forall (d,u,v) \in DU(P), \forall r \in R_{v,u}, \exists \pi \in T : \pi \vDash \varphi_{d,u,v,r}.$$

The coverage requirement for a single range r can also be expressed as:

$$\varphi_{d,u,v,r} = F(\texttt{s.loc} = u \wedge \texttt{s.v.def} = d \wedge ((\exists\, r' \in \texttt{ranges(s.v.constraint)} : r' \cap r \neq \varnothing)),$$

where `s` represents a symbolic program state and `ranges(s.v.constraint)` denotes the set of value ranges implied by symbolic constraints on `v` at that state as defined in section 4.1.

This LTL query enables the use of standard model checking techniques to search for executions that exercise the specified def–use pair under the given value-range constraints. While the above formulation assumes a fixed target range $r \in R_{v,u}$, the same mechanism can be extended to support dynamic, on-the-fly discovery of uncovered value ranges. Let `RANGES` denote the set of value ranges already covered for the selected def–use pair. Initially, no ranges are considered covered (or only those exercised by an existing test suite). At each iteration, the model checker is queried with:

$$F(\texttt{s.loc} = u \wedge \texttt{s.v.def} = d \wedge ((\exists\, r \in \texttt{ranges(s.v.constraint)} : r \setminus \texttt{RANGES}(d,u,v) \neq \varnothing)).$$

This query effectively asks whether there exists a feasible execution in which the definition–use association (`d`,`u`,`v`) is exercised and the value of `v` at the use location `u` falls into a value range that has not yet been covered. If the formula holds, the model checker produces a counterexample path, which directly yields a new test case. From the symbolic constraints associated with this path, one or more previously uncovered value ranges are extracted and merged into `RANGES`. The process is then repeated with the refined set of covered ranges.

The procedure terminates when the corresponding LTL query yields no counterexample, i.e., when the model checker determines that no feasible execution exists that reaches the selected def–use pair with a value range not yet covered. At this point, all distinct value ranges that can feasibly reach the use location

have been exercised. Note that loops do not introduce additional coverage obligations once a fixpoint in the value ranges is reached.

This iterative refinement process is conceptually similar to counterexample-guided abstraction refinement [22, 23] (CEGAR): the set of already covered value ranges may be considered as a current abstraction, the LTL query checks for the existence of a counterexample 'violating' the current coverage state, and each counterexample is used to refine the abstraction by introducing a new value range. However, unlike traditional CEGAR approaches that refine control-flow or predicate abstractions, our method refines value partitions directly driven by the RS-DFC requirements. Moreover, termination of the procedure is guaranteed:

- refinements are monotone since each refinement may only extend the accumulated maximal value ranges stored in `RANGES`;
- since the exploration considers a finite set of symbolic states modulo constraint equivalence, only finitely many distinct maximal value ranges can be induced at each use location. Once the maximal range associated with a def–use pair has been reached, further exploration cannot produce new refinements.

Yet this approach avoids the need to enumerate all def–use–range combinations in advance, while ensuring that all distinct value ranges reaching the use location are eventually covered.

Discussion. The approaches provide systematic test generation, ensuring that all reachable value ranges for each def–use pair are represented and exercised. This enables the derivation of semantically richer test cases and is particularly valuable in safety-critical applications and input validation scenarios, where untested ranges may lead to serious faults. Importantly, the outcome of the proposed approach is not limited to a test suite. The algorithms also explicitly construct the set of admissible value ranges reaching each computation use, thereby characterizing the reachable value domain – even when a single range is sufficient to meet the coverage criterion.

However, these approaches are subject to the well-known limitation of state space explosion [6, 8, 17, 24], which can restrict scalability when analyzing large programs or employing fine-grained range partitions. Despite this, they offer a favorable trade-off between computational cost and the increased reliability gained through sensitivity to the values being tested.

5. Related work

Reasoning about value ranges and boundary conditions has long been recognized as a fundamental aspect of effective software testing [1, 19]. Classical black-box test design techniques, such as equivalence partitioning and boundary value analysis, systematically partition the input domain into equivalence classes and emphasize testing at domain boundaries, where faults are more likely to manifest [2]. For ordered input domains, representative values at and near extreme points are selected to expose discontinuities, off-by-one errors, and threshold-related faults. These techniques provide an early conceptual foundation for treating value partitions as explicit testing objectives.

In white-box testing, structural coverage criteria shift the focus from input domains to program structure. Dataflow-based criteria – including all-uses and def–use chains coverage – exercise the relationships between definitions and uses of variables along control-flow paths [3, 4, 13, 25]. While these criteria ensure that data dependencies are structurally exercised, they abstract away from the concrete values propagated along these paths. As a result, test suites satisfying traditional def–use coverage may still fail to exercise variativity of distinct value contexts at computation uses, even when those values induce substantially different program behaviors [15].

Several works on dataflow test generation address feasibility by employing symbolic execution to construct inputs that satisfy def–use requirements [8, 15]. Symbolic and concolic execution [6, 7, 17, 21] techniques reason about program inputs symbolically and accumulate path conditions to generate concrete test cases via constraint solving. Although these approaches can distinguish executions that follow the same control-flow path under different constraints, they are primarily driven by structural coverage objectives.

In parallel, the static analysis community [9, 26] has developed mature techniques for value range analysis, typically framed as abstract interpretation over domains such as intervals, finite sets, or relational abstractions. Such analyses compute sound over-approximations of the values that variables may take at specific program points and are widely used in compilers and verification tools for optimization, runtime error detection, and specification checking. Platforms such as Frama-C [9] provide value analysis plug-ins that infer variable ranges and support advanced analyses through path sensitivity and predicated abstract domains. Despite their precision and practical relevance, these analyses are primarily used for verification and diagnostics rather than as explicit coverage criteria guiding test generation.

More recent research has explored integrating semantic information into testing objectives. Symbolic execution engines often incorporate heuristics to prioritize boundary conditions of path constraints, effectively probing value extremes to improve fault detection. Other approaches investigate specification-based or feature-based partitioning, defining coverage goals over high-level behavioral features or specified correctness properties [27]. While these methods acknowledge the importance of semantic distinctions, they typically focus on input-level properties or specific application domains and do not formalize coverage of value partitions in relation to def–use pairs relationships [1–4, 20, 26].

Range-sensitive extensions of dataflow testing have been proposed to bridge this gap. Our work explicitly treats value ranges as first-class coverage objectives. Unlike classical dataflow criteria, RS-DFC requires that, for each targeted def–use pair, all logically distinct value ranges that can reach the use along feasible paths are exercised. In contrast to static value analysis used solely for verification, RS-DFC leverages path-sensitive constraints to define concrete coverage goals for test generation. Finally, RS-DFC is well-suited for cyber-physical systems, as it combines path-sensitive exploration with systematic coverage of value ranges reaching actuator and sensor interfaces [11, 12, 28], capturing value-dependent behaviors driven by continuous physical signals and thereby increasing confidence in correct system operation.

Results. We introduce the Range-Sensitive Data-Flow Coverage (RS-DFC) criterion, a novel approach that extends classical dataflow coverage by distinguishing values reaching each def–use pair. To achieve this in practice, we developed two complementary algorithmic approaches. The first relies on symbolic state-space exploration, dynamically generating test objectives while incrementally discovering and exercising previously unobserved value ranges along feasible paths. The second encodes RS-DFC obligations as Linear Temporal Logic formulas, allowing model checking to systematically identify executions that cover value ranges not yet exercised. Both approaches produce coverage paths along with the associated set of covered value ranges, capturing the extent of admissible value variation exercised by each test. Through illustrative examples, we showed how RS-DFC can refine conventional coverage criteria: test suites satisfying branch, all-uses, or MC/DC criteria may overlook behaviors associated with distinct value partitions, which RS-DFC explicitly captures. This demonstrates that systematically integrating value sensitivity can uncover faults missed by purely structural coverage while remaining compatible with established testing practices.

Conclusions. RS-DFC offers a more precise measure of test adequacy in scenarios where observable program behavior depends on the values propagated along def–use paths. Its benefits are most pronounced for input-related definitions and critical computation sinks, where value variability directly influences correctness or safety. Rather than replacing existing coverage criteria, RS-DFC complements structural approaches by highlighting value-dependent behaviors that conventional metrics overlook. The criterion is particularly applicable to the systematic testing of input validation, boundary handling, and data-dependent decision logic, enabling the discovery of subtle faults that could remain undetected under classical all-uses or branch coverage. The proposed approach can be effectively employed as an auxiliary testing technique for cyber-physical systems, with a particular focus on input-sensitive definitions arising from sensor readings, actuator states, and intermediate computations. This enables systematic detection of weak input validation and unsafe state transitions, which may give rise to potential cybersecurity or safety vulnerabilities.

However, achieving full RS-DFC can incur significant computational overhead, especially in programs with large or complex symbolic state spaces, and therefore may not be justified for every variable or location. To further improve scalability without compromising coverage, it is reasonable to consider heuristic-guided state-space traversal, hybrid testing strategies, and the integration of existing static analysis techniques.

Authorship contributions: Oleksandr Kolchyn – conceptualization and formalization of the approach; Daria Rudenko – investigation and description of test generation methods, development of examples, experimental evaluation.

Data Availability. The data used in this study are derived from publicly available sources and are cited in the corresponding bibliographic references. The reported results and conclusions are based on formal analysis and theoretical reasoning.

Funding. This research was conducted without external funding.

References

1. Homès B. Fundamentals of software testing. John Wiley & Sons. 2024. https://doi.org/10.1002/9781118602270
2. Herrera A., Payer M., Hosking A. DatAFLow: Toward a Data-Flow-Guided Fuzzer. *ACM Transactions on Software Engineering and Methodology*. 2023. **32**(5). 31 p. https://doi.org/10.1145/3587156
3. Frankl P., Weyuker E. J. An Applicable Family of Data Flow Testing Criteria. *IEEE Transactions on Software Engineering*. 1988. **14** (10). P. 1483–1498. https://doi.org/10.1109/32.6194
4. Su T. et al. A survey on data-flow testing. ACM Comput. Surv. 2017. Vol. 50. art. 5. 35 p. https://doi.org/10.1145/3020266
5. Gay G., Staats M., Whalen M., Heimdahl M. The risks of coverage-directed test case generation. *IEEE Transactions on Software Engineering*. 2015. Vol. 41. P. 803–819. https://doi.org/10.1109/TSE.2015.2421011
6. Kurian E. et al. Automatically generating test cases for safety-critical software via symbolic execution. *Journal of Systems and Software*. 2023. 111629. https://doi.org/10.1016/j.jss.2023.111629
7. Wang X., Sun J., Chen Z., Zhang P., Wang J., Lin Y. Towards optimal concolic testing. In Proceedings of the 40th Int. Conf. on Software Engineering. 2018. P. 291–302. https://doi.org/10.1145/3180155.3180177
8. Su T. et al. Towards Efficient Data-Flow Test Data Generation. In: Bowen, J.P., Li, Q., Xu, Q. (eds) Theories of Programming and Formal Methods. *Lecture Notes in Computer Science*. 2023. Vol. 14080. Springer, Cham. https://doi.org/10.1007/978-3-031-40436-8_10
9. Kosmatov N., Prevosto V., Signoles J. Guide to Software Verification with Frama-C. Springer. 2024. https://doi.org/10.1007/978-3-031-55608-1
10. Chilenski J., Miller S. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*. 1994. Vol. 7, No. 5. P. 193–200, https://doi.org/10.1049/sej.1994.0025
11. Letichevsky A., Letychevskyi O., Skobelev V., Volkov V. Cyber-physical systems. *Cybernetics and Systems Analysis*. 2017. 53(6). P. 821–834. https://doi.org/10.1007/s10559-017-9984-9
12. Sadri-Moshkenani Z., Bradley J., Rothermel G. Survey on test case generation, selection and prioritization for cyber-physical systems. Softw Test Verif Reliab. 2022. 32: e1794. https://doi.org/10.1002/stvr.1794
13. Rapps S., Weyuker E. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on software engineering*. 1985. Vol. 4. https://doi.org/10.1109/TSE.1985.232226
14. Lengauer T., Tarjan R. A fast algorithm for finding dominators in a flowgraph. *ACM Trans Program Lang Syst*. 1979. **1** (1). P. 121–141. https://doi.org/10.1145/357062.357071
15. 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. *Lecture Notes in Computer Science*. 2022. Vol 13274. Springer, Cham. https://doi.org/10.1007/978-3-031-07727-2_17
16. Paul S. et al. Formal Verification of Safety-Critical Aerospace Systems. *IEEE Aerospace and Electronic Systems Magazine*. 2023. Vol. 38, No. 5. P. 72–88. https://doi.org/10.1109/MAES.2023.3238378
17. Cadar C., Sen K. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM*. 2013. Vol. 56, Iss. 2. P. 82–90. https://doi.org/10.1145/2408776.2408795
18. Clarke E., Henzinger T., Veith H., Bloem R. (Eds.). Handbook of model checking. 2018. Vol. 10. P. 978–3. Cham: Springer. https://doi.org/10.1007/978-3-319-10575-8
19. Myers Glenford J. et al. The art of software testing. 2004. Vol. 2. Chichester: John Wiley & Sons.

20. Potiyenko S., Kolchyn O. Model-based automated refurbishing of test input data. *IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*. Montreal, QC, Canada. 2025. P. 198–202. https://doi.org/10.1109/SANER-C66551.2025.00036
21. King J. C. Symbolic Execution and Program Testing. *Communications of the ACM*. 1976. https://doi.org/10.1145/360248.360252
22. Clarke E., Grumberg O., Jha S., Lu Y., Veith H. Counterexample-Guided Abstraction Refinement . In: Emerson E., Sistla A. (eds) Computer Aided Verification (CAV 2000). *Lecture Notes in Computer Science*. 2000. Vol. 1855. Springer, Berlin, Heidelberg. https://doi.org/10.1007/10722167_15.
23. Kreft R., Büchner C., Sievers S., Helmert M. Computing Domain Abstractions for Optimal Classical Planning with Counterexample-Guided Abstraction Refinement. *Proceedings of the International Conference on Automated Planning and Scheduling*. 2023. **33** (1). P. 221–226. https://doi.org/10.1609/icaps.v33i1.27198
24. Kolchyn O., Potiyenko S. Model-Based Test Cases Generation for Extended Data Flow Coverage Criteria. In: Silhavy R., Silhavy P. (eds) Software Engineering Methods Design and Application. CSOC 2024. *Lecture Notes in Networks and Systems*. 2024. Vol. 1118. Springer, Cham. https://doi.org/10.1007/978-3-031-70285-3_43
25. Hong H., Ural H. Dependence Testing: Extending Data Flow Testing with Control Dependence. In: Khendek F., Dssouli R. (eds) Testing of Communicating Systems (TestCom 2005). *Lecture Notes in Computer Science*. 2025. Vol. 3502. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11430230_3
26. Sellami Y., Girol G., Recoules F., Couroussé D., Bardin S. Inference of robust reachability constraints. *Proceedings of the ACM on Programming Languages*. 2024. Vol. 8. P. 2731–2760. https://doi.org/10.1145/3632933
27. Al Blwi, et al. Semantic Coverage: Measuring Test Suite Effectiveness. In Proceedings of the 18th International Conference on Software Technologies (ICSOFT). 2023. P. 287–294. https://doi.org/10.5220/0012063900003538
28. Samoliuk T.A. Embedded Systems Technologies Using IoT and Wireless Sensor Networks in Semi-Real-Time Modeling. *Cybernetics and Computer Technologies*. 2025. 1. P. 98–105. (in Ukrainian) https://doi.org/10.34229/2707-451X.25.1.10

Received/Одержано 13.02.2026 Accepted/Прийнято 26.05.2026 Published/Надруковано 01.06.2026