STRUST-v1: Slice-Based Differential Verification for Large COBOL-to-Java Migrations

George Weale
Columbia University
New York, NY, USA
george.weale@columbia.edu

***Abstract*—Modernizing business-critical COBOL workloads to Java creates a verification problem that is more subtle than compilation success or source-level similarity: the converted program must preserve the observable behavior on which downstream operations depend. This paper presents STRUST, a contract-driven differential-verification framework that scales from a single program to a staged business flow. The key abstraction is a *verification slice*: a context-bounded, executable closure of the programs, copybooks, data contracts, fixtures, boundary adapters, and observations needed to validate one business-visible outcome. STRUST derives slices from a system dependence graph and observed execution traces, then recomposes them through progressively larger stages: record, program, call chain, job, and end-to-end flow. A test contract fixes source revisions, runtime configuration, fixtures, environmental adapters, normalization policy, and comparator semantics before either implementation is executed. The framework emits a tri-state verdict—MATCH, DIVERGE, or INCONCLUSIVE—and retains raw evidence at every stage. The version-1 "Verified Snippet Demonstrator" is evaluated on ten controlled scenarios. Its supplied result set yields 80% classification accuracy and a mismatch-detection F1 score of 0.75; the two failures expose precisely the normalization and decimal-semantics issues that a credible modernization verifier must surface. The contribution is not a claim that containers solve mainframe emulation, but a reproducible method for assembling small, inspectable equivalence claims into a larger migration-evidence case.**

***Index Terms*—legacy modernization, COBOL, Java, differential testing, program slicing, interprocedural analysis, call graph, test oracle**

I. INTRODUCTION

COBOL-to-Java modernization is a semantic preservation problem. A conversion can compile cleanly, satisfy a style checker, and still be operationally unsafe if it changes a rounding boundary, a field-padding rule, a collation order, an error path, or the state written to a downstream system. These hazards are unsurprising: COBOL data descriptions and operational environments encode behavior that is often implicit in a Java translation. In particular, fixed-point decimal fields, packed-decimal representations, EBCDIC text, record-oriented I/O, and dialect-specific runtime behavior require an explicit compatibility decision rather than a syntactic rewrite [5], [6].

Traditional modernization assurance combines unit tests, integration tests, code review, and vendor-specific checks. Those practices remain necessary, but none independently answers a narrow and important question: *given the same approved workload and declared environment, did the candidate preserve the reference program's observable behavior?* Differential testing provides a useful foundation because independently executed implementations can act as mutual test oracles [1]. Its value, however, depends on disciplined control of inputs, environments, and comparison semantics.

This work introduces STRUST (*semantic trust*), an evidence-oriented framework for differential verification of a COBOL reference program and a Java candidate. It deliberately avoids a stronger claim that finite testing proves equivalence across all inputs. Instead, STRUST reports agreement over an explicit workload under an explicit contract; disagreement becomes a localized artifact with retained evidence.

The paper makes five contributions:

1) a contract model that makes environmental assumptions and equivalence semantics first-class artifacts;
2) a containerized architecture with reproducible execution capsules and an auditable trace ledger;
3) a slice-based scale-out method that connects business observations to context-aware call and data dependencies;
4) a staged recomposition plan for verifying large batch and transaction flows without treating the entire estate as one opaque test; and
5) a candid evaluation of the version-1 demonstrator, including corrected diagnostic metrics and failure analysis.

II. PROBLEM MODEL AND DESIGN PRINCIPLES

A. Observable Equivalence Is Contractual

For every paired execution, STRUST captures a trace bundle: exit classification, standard streams, declared output files, declared persistent-state deltas, timeout state, and run provenance. The trace is intentionally broader than console output. A program that prints the expected line but writes the wrong record layout, database row, or return condition has not preserved the business behavior that matters.

A *test contract* fixes the adapters, canonicalization policy, comparator, resource limits, and observation scope. Exact-byte comparison is appropriate for an approved binary artifact; it is not a defensible default when one side emits EBCDIC and the other UTF-8. Conversely, unconstrained whitespace, timestamp, or numeric "normalization" can hide a real defect. Every transformation must therefore be declared, versioned, and attributable to a compatibility decision.

B. Three Verdicts, Not a Forced Boolean

STRUST classifies each fixture as MATCH, DIVERGE, or INCONCLUSIVE. A MATCH means that all required obser-

vations compare equal after authorized normalization. A DIVERGE means that both executions reached a comparable terminal state and at least one required observation differs. An INCONCLUSIVE result represents a timeout, compilation failure, missing adapter, unapproved dependency, or incompatible environmental precondition. This separation prevents a harness failure from being misreported as either a functional mismatch or a successful conversion.

C. Evidence, Not a Magic Number

STRUST reports agreement rate, divergence rate, inconclusive rate, covered business observations, contract identity, and dependency frontier. A future deployment can weight these by business risk, but the v1 paper deliberately avoids burying uncertainty in a single score. High agreement is evidence for the enumerated workload—not a proof that every untested path is equivalent. This distinction aligns with the long-standing limits of test adequacy and coverage criteria [2], [3].

III. System Architecture

Figure 1 shows the architecture. The design centers on the contract manifest rather than the containers: container isolation is useful only when the runtime identity, fixture materialization, and comparison policy are reproducible.

A. Contract Manifest

The contract is a version-controlled manifest with six required sections:

1) *Identity*: SHA-256 digests for source, copybooks, build files, images, and comparator plug-ins.
2) *Execution*: compiler/JDK versions, dialect flags, locale, timezone, environment variables, CPU and wall-clock limits, and network policy.
3) *Fixtures*: immutable input files, command-line arguments, file allocations, and declared initial state.
4) *Adapters*: mappings for file paths, encodings, external calls, and simulated subsystems.
5) *Observation*: streams, files, return conditions, state snapshots, and any interaction trace that must be compared.
6) *Comparison*: canonicalization transformations, schema-aware comparators, tolerances, and forbidden normalizations.

This structure makes the usual hidden assumptions inspectable. For example, a contract can state that a DISPLAY field is transcoded from EBCDIC to UTF-8 before textual comparison, while a COMP-3 field is decoded by a field-aware adapter rather than treated as text. IBM documents packed decimal as a storage representation distinct from DISPLAY and NATIONAL data; a generic character transcoder is therefore not an adequate comparator for such data [5].

B. Execution Capsules and Trace Capture

The prototype uses separate, pinned container images for the COBOL and Java toolchains. The COBOL capsule can use GnuCOBOL for portable demonstration cases [6]; production contracts should name the actual compiler, dialect options, and runtime compatibility layer. The Java capsule pins the JDK, dependency lockfile, and relevant JVM options. Neither capsule receives mutable test input from the other.

Algorithm 1 Contract-driven differential verification

Require: Contract Π, fixture x, reference P_C, candidate P_J
$\Pi \leftarrow$ VALIDATEANDDIGEST(Π)
$(e_C, e_J) \leftarrow$ BUILDPINNEDCAPSULES(Π)
$(o_C, o_J) \leftarrow$ EXECUTEINPARALLEL(P_C, P_J, x, e_C, e_J)
if NOTCOMPARABLE(o_C, o_J, Π) **then**
  return (INCONCLUSIVE, PERSISTRAWEVIDENCE())
end if
$(\hat{o}_C, \hat{o}_J) \leftarrow (N_\Pi(o_C), N_\Pi(o_J))$
$d \leftarrow C_\Pi(\hat{o}_C, \hat{o}_J)$
$v \leftarrow$ **if** d **then** MATCH **else** DIVERGE
return (v, PERSISTEVIDENCE($\Pi, o_C, o_J, \hat{o}_C, \hat{o}_J$))

Each run emits a content-addressed trace bundle. At minimum, it contains the manifest digest, source and image digests, invocation, exit class, captured streams, declared output artifacts, timing metadata, and comparator version. The collector writes raw evidence before normalization. This ordering is important: a later change to canonicalization policy must not destroy the ability to inspect what each program actually produced.

C. Canonicalization and Comparison

Canonicalization is a compatibility boundary, not a cosmetic clean-up step. STRUST distinguishes:

- *representation normalization*, e.g., CRLF-to-LF conversion or a declared EBCDIC code-page transcode;
- *schema normalization*, e.g., decoding a fixed-length record into named fields before comparison; and
- *semantic comparison*, e.g., an approved decimal scale and rounding mode, or order-insensitive comparison of an explicitly unordered collection.

The engine applies the narrowest comparator consistent with the contract: byte comparison for invariant bytes, record-aware comparison for mainframe files, structured comparison for JSON/XML/CSV, and state comparison for declared persistence adapters. A transformation that masks a business-visible difference is rejected as a contract violation. Timestamp stripping, for example, is permitted only for a named field that has been classified as non-semantic.

IV. Scaling from Snippets to Business Flows

A. The Verification Slice

A large COBOL estate cannot be verified credibly by either extreme: running disconnected toy snippets forever, or attempting to boot the entire enterprise as one opaque test. STRUST uses a middle abstraction, the *verification slice*. A slice is not an arbitrary source fragment. It is the smallest context-bounded execution unit that can establish one named business observation, together with the dependencies and environmental assumptions required to make that observation meaningful.

For example, an observation might be "the settlement output record for a rejected payment," "the updated balance field after

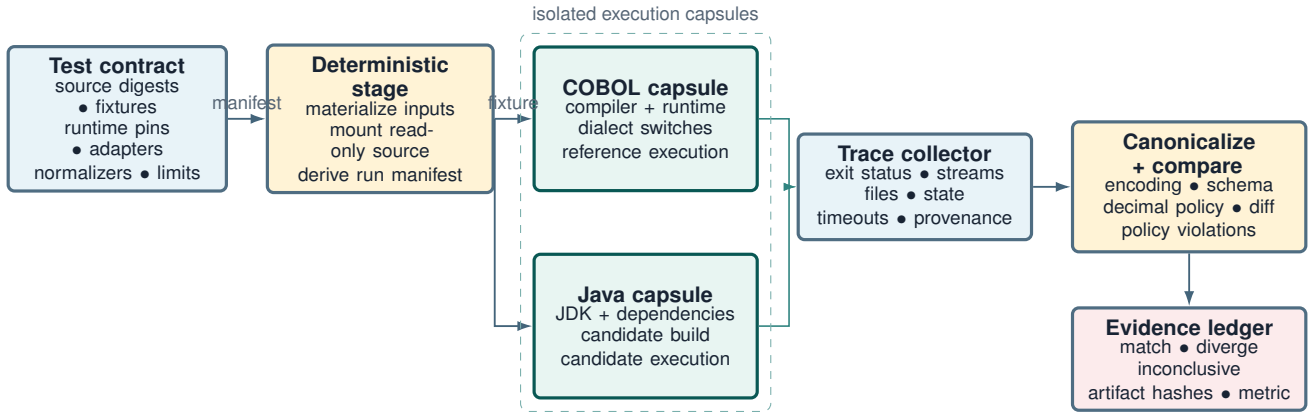

Fig. 1. STRUST architecture. The test contract governs every comparison-relevant choice. Containers isolate execution; the evidence ledger retains the information needed to reproduce or challenge a verdict.

TABLE I
STAGED VERIFICATION LADDER. EACH HIGHER STAGE REUSES THE EVIDENCE AND CONTRACTS FROM ITS LOWER STAGES.

Stage	Verification unit	Primary evidence
Record	Named field or record layout	Raw bytes, encoding, scale, sign, schema.
Program	One compilation unit and its paragraphs	Return condition, files, streams, selected storage.
Chain	Context-specific program/call path	Parameter mapping, interaction trace, call context.
Job	Batch steps and allocated datasets	Step ordering, dataset state, restart/error behavior.
Flow	End-to-end business scenario	Cross-boundary state transitions and release evidence.

interest accrual," or "the return condition and journal record produced by a batch step." Starting from that observation, the slice includes the relevant COBOL paragraphs or programs, copybooks, file layouts, input records, initialized storage, call context, and boundary adapters. A slice may be small, but it remains executable; it is a testable piece of a system rather than a code excerpt pasted into a harness.

This ladder lets a team validate the hard, high-risk parts first and then assemble the evidence upward. A green program slice does not automatically make a green batch job; the job-level contract adds JCL parameters, allocated datasets, step ordering, restart behavior, and external boundary conditions. Conversely, when a flow diverges, the retained lower-stage evidence makes the failure localizable instead of forcing a blind end-to-end debugging exercise.

B. Discovering Call and Data Context

The planner begins with a repository inventory: COBOL programs, copybooks, JCL, configuration, file layouts, database schemas, and known subsystem contracts. It constructs a unified dependency graph with control edges (paragraph `PERFORM`, static `CALL`, transaction transfer, and job-step sequencing), data edges (record fields, files, tables, and shared storage), and build edges (copybooks, compiler options, and linked modules). A program dependence graph exposes data and control dependences; its interprocedural extension supplies the right model for crossing procedure boundaries [10], [11]. Program slicing then lets a business observation be traced backward through the dependencies that can affect it [9].

For a modern engineer, "find the function's call stack" must be translated carefully for COBOL. The useful targets may be a paragraph, section, entry point, subprogram, CICS transfer, or JCL `EXEC` step. An out-of-line `PERFORM` is an intra-program control relationship; a `CALL` or transaction transfer may cross a program boundary; a batch flow may cross a scheduler boundary. STRUST records a *context key*—job or transaction identity, program identifier, normalized call path, fixture version, and declared dataset state—so that the same callee reached from two business paths is not incorrectly treated as one test context.

Static analysis will not resolve every edge. Dynamic program names, indirect calls, generated JCL, runtime configuration, and external services can leave the graph incomplete. For those cases, the planner imports observed edges from reference execution traces: opened datasets, resolved module names, transaction identifiers, boundary invocations, and relevant call-path events. An unobserved or unresolved dependency is surfaced as a frontier item, not silently omitted from a slice.

C. Boundary Control and Bytewise Evidence

The point where a slice meets the rest of the system is its *boundary frontier*. Each frontier edge is classified before execution: (i) included in the slice, (ii) represented by a deterministic adapter, (iii) observed but not yet modeled, or (iv) blocked. Only the first two states permit a MATCH verdict. The third produces INCONCLUSIVE evidence with an explicit follow-up item; the fourth blocks the slice. This rule prevents a convenient mock from masquerading as a verified integration.

Bytewise comparison is a powerful gate, but only at the correct representation layer. For a fixed record with identical code page, layout, and semantic scope, STRUST compares the raw artifact byte for byte and retains the exact offset range of a difference. If the COBOL and Java sides legitimately use different physical encodings, the framework first decodes both

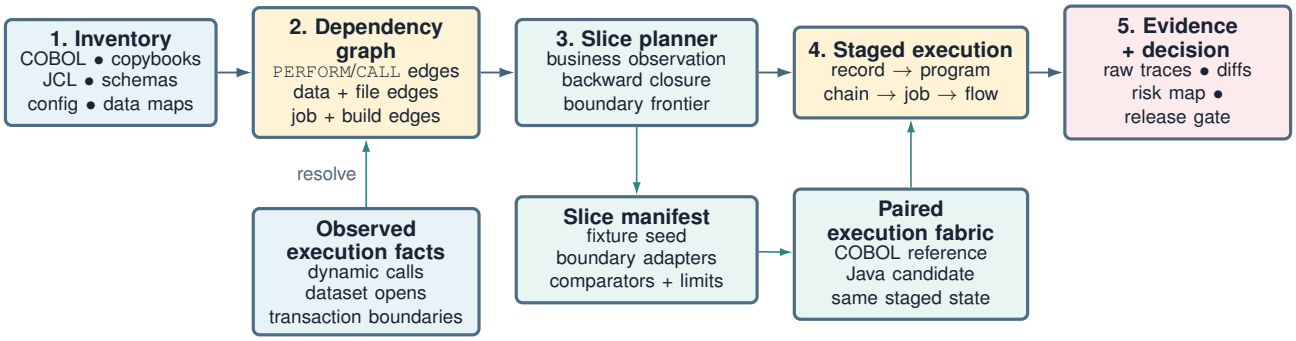

Fig. 2. System-scale verification flow. STRUST turns a large codebase into context-aware verification slices, executes them through a staged ladder, and retains observed dependencies and missing fixtures as planning inputs.

Algorithm 2 Slice planning and staged recomposition

Require: Inventory I, business observations Z, contract templates Π

$G \leftarrow \textsc{MergeEdges}(I)$
$Q \leftarrow [\,]$ ▷ planned verification slices
for all $z \in Z$ **do**
  $S \leftarrow \textsc{SliceClosure}(G, z)$
  $B \leftarrow \textsc{BoundarySet}(S, G)$
  $K \leftarrow \textsc{BindContract}(S, B, \Pi)$
  $Q \leftarrow Q \cup \{\textsc{Slice}(S, B, K, z)\}$
end for
$L \leftarrow \textsc{TopologicalStages}(Q)$
for all stage $\ell \in L$ **do**
  for all slice q in ℓ **do**
    $v \leftarrow \textsc{Verify}(q)$
    if $v \in \{\textsc{Diverge}, \textsc{Inconclusive}\}$ **then**
      $\textsc{PersistBlock}(q, v)$
    end if
  end for
end for
return $\textsc{ReleaseEvidence}(Q, L)$

artifacts through a contract-pinned schema into a canonical record model, compares the named fields and decimal rules, and preserves the original bytes alongside the result. This is how the framework can say both "the generated bytes changed at offset 42" and "the business amount diverged after a COMP-3 decode" without confusing a transport conversion with a functional defect.

D. Context-Preserving Construction and Recomposition

The scale-out algorithm in Algorithm 2 creates slices from a set of business observations rather than from a directory of source files. Its central safety property is monotonic evidence: a larger stage may add behavior and dependencies, but it may not discard a lower-stage divergence, contract violation, raw trace, or unresolved boundary. A failed leaf slice blocks recomposition until it is fixed, explicitly waived as a behavioral change, or reclassified after the contract is corrected.

V. Version-1 Demonstrator

A. Scope

The "Verified Snippet Demonstrator" is a deliberately constrained proof of concept. It targets self-contained programs using arithmetic, conditional logic, iteration, line-oriented input, sequential output, and standard streams. It uses a Python orchestration layer and Docker to build, execute, collect, normalize, compare, and report. The COBOL side uses GnuCOBOL 3.1.2 and the Java side uses OpenJDK 11, matching the supplied v1 test environment. The prototype does *not* yet implement the system dependence graph, dynamic call-trace collection, slice planner, CICS/IMS/DB2/VSAM adapters, JCL-driven allocation, or production mainframe runtime emulation described in Section IV.

The scope boundary is a feature, not a footnote. Treating an unsupported subsystem as an ordinary mismatch would pollute results; treating it as a match would be unsafe. In STRUST it is a contract failure or an INCONCLUSIVE verdict until an adapter and observation policy are supplied.

B. Decimal Semantics: A Representative Failure Mode

Financial code exposes why source-level similarity is insufficient. A COBOL field such as `PIC 9(6)V99` denotes a fixed decimal scale in the program model; a Java `double` is a binary floating-point value. Formatting a `double` at the end of a calculation does not restore the intermediate decimal semantics. The Java platform's `BigDecimal` API exists specifically for arbitrary-precision decimal arithmetic and explicit rounding behavior [7].

Listing 1 illustrates the candidate-side discipline required by a contract that specifies a two-decimal stored scale and `HALF_UP` rounding. The substring offsets, scale, denominator, and rounding mode are not implementation trivia: each is part of the equivalence surface and must be derived from the source record definition and contract.

VI. Evaluation

A. Methodology

The v1 supplied test suite contains ten manually constructed COBOL/Java pairs. It includes arithmetic, string handling, conditionals, iteration, sequential file I/O, numerical precision,

Listing 1. Contract-aligned fixed-point calculation in Java.

```
BigDecimal amount = new BigDecimal(line.substring(0, 8)).
     movePointLeft(2);
BigDecimal rate = new BigDecimal(line.substring(8, 12)).
     movePointLeft(2);
BigDecimal interest = amount.multiply(rate).multiply(
     BigDecimal.valueOf(term))
   .divide(BigDecimal.valueOf(100), 2, RoundingMode.HALF_UP)
        ;
```

order dependence, error handling, rounding behavior, and character encoding. Five cases were designed to agree; four were deliberately non-equivalent; and one EBCDIC case was intended to agree after normalization. This is an artifact-level evaluation, not a benchmark of conversion systems and not a scalability study.

For diagnostic calculations, a *detected mismatch* is treated as the positive class. Thus, a true positive is a genuinely non-equivalent pair reported as DIVERGE; a false positive is an equivalent pair reported as DIVERGE; and a false negative is a non-equivalent pair reported as MATCH. We report accuracy, precision, recall, and F1 because a raw pass rate can conceal whether the comparator is failing open or failing closed.

B. Results

Table II yields the confusion matrix in Table III: $TP = 3$, $FP = 1$, $FN = 1$, and $TN = 5$. The demonstrator correctly classifies eight of ten cases, giving an accuracy of 0.80. For the mismatch class, precision, recall, and F1 are each 0.75; specificity is $5/6 \approx 0.83$. The Wilson 95% interval for the 80% accuracy estimate is wide ($[0.49, 0.94]$), which is expected with only ten scenarios. The appropriate conclusion is feasibility with known defects, not a production-level accuracy claim.

The two failures are technically informative. T9 demonstrates that tolerance policies must never silently override source-defined decimal scale and rounding rules. A converter may compute a financially material value incorrectly while remaining within an arbitrary epsilon. T10 demonstrates that byte-wise or Unicode-only text comparison is insufficient when the reference contract permits EBCDIC data. The remedy is not to disable comparison; it is to use a declared code-page and field-schema adapter, preserving raw bytes in the evidence bundle.

C. Scale-up Evaluation Plan

The proposed slice architecture requires a separate evaluation; the ten-case demonstrator does not validate it. A credible system-scale study should report at least five dimensions. *Closure* measures the percentage of required static and observed dependencies either included or explicitly modeled at the frontier. *Recomposition yield* measures how many green lower-stage slices remain green when assembled into a chain, job, or business flow. *Localization quality* measures the distance from an injected or discovered divergence to the first failing slice. *Cost* measures wall-clock time, parallel capacity, container reuse, and evidence-storage growth. Finally, *diagnostic utility* measures whether the retained trace, record diff, and context key let an engineer identify the responsible conversion boundary without rerunning the entire workload.

The study should preserve both reference and candidate artifacts, preregister the comparator policy, include deliberately seeded faults at record, call-boundary, and job-order levels, and separately label behavior changes approved by domain owners. The test-oracle literature makes clear why an old implementation is evidence rather than an infallible specification; a modernization program still needs an explicit mechanism for accepting intentional corrections [12].

VII. DISCUSSION

A. What STRUST Can Establish

STRUST can establish reproducible behavioral agreement for the tested workload under the declared contract. It is valuable as a modernization quality gate because it produces evidence that can be inspected by developers, auditors, vendors, and risk owners. It can also compare different conversion candidates fairly, provided they are evaluated against the same corpus, contract, and comparator policy.

The framework cannot establish that a legacy program is correct, that every possible input is covered, or that a Java program is universally equivalent to its COBOL counterpart. The reference program may contain a latent defect; a conversion may intentionally repair it. That scenario deserves a signed behavioral-change exception, not an unannotated test failure or a quietly altered normalizer. Similarly, a green result for a GnuCOBOL capsule is not evidence of compatibility with an arbitrary z/OS configuration. Compiler dialect, NUMPROC behavior, collation, and subsystem semantics must be encoded in the contract or reported as limitations.

B. Positioning with Other Assurance Techniques

Differential verification complements rather than replaces static analysis, human review, property-based testing, security assessment, and integration testing. Static analysis is useful for dependency discovery and unsupported construct detection; differential execution is useful for detecting runtime semantic drift. Coverage instrumentation can prioritize fixtures, but coverage alone does not guarantee fault detection [2], [4]. The strongest modernization program uses these techniques together and treats STRUST's evidence as one decision input rather than a universal oracle.

VIII. LIMITATIONS AND THREATS TO VALIDITY

External validity. Ten small, manually prepared snippets do not represent enterprise call graphs, batch schedules, or stateful transaction systems. The reported metrics should not be generalized to a vendor, a language model, or a production portfolio.

Oracle validity. Differential comparison assumes that the COBOL behavior is the desired reference behavior. Known legacy defects require an explicit exception mechanism; otherwise, a correct repair appears as a divergence.

Environment validity. GnuCOBOL is appropriate for an accessible demonstration but is not a substitute for every Enterprise COBOL dialect or z/OS subsystem. CICS, IMS, DB2,

TABLE II
VERSION-1 RESULTS, RECONSTRUCTED FROM THE SUPPLIED TEST LEDGER. "CORRECT" EVALUATES THE DEMONSTRATOR VERDICT AGAINST THE INTENTIONALLY ASSIGNED GROUND TRUTH.

ID	Scenario	Ground truth	Verdict	Correct	Diagnostic interpretation
T1	Arithmetic	Equivalent	MATCH	Yes	Baseline fixed-operation agreement.
T2	String processing	Equivalent	MATCH	Yes	Concatenation and substring behavior agreed.
T3	Conditional logic	Equivalent	MATCH	Yes	Selected branch outcomes agreed.
T4	Iteration	Equivalent	MATCH	Yes	`PERFORM VARYING` translation agreed.
T5	Sequential file I/O	Equivalent	MATCH	Yes	Declared output files compared equal.
T6	Numerical precision	Non-equivalent	DIVERGE	Yes	Detected decimal/binary precision divergence.
T7	Order dependence	Non-equivalent	DIVERGE	Yes	Detected sequence-sensitive output difference.
T8	Error handling	Non-equivalent	DIVERGE	Yes	Detected different failure behavior.
T9	Rounding behavior	Non-equivalent	MATCH	No	False negative; minor rounding difference escaped policy.
T10	Character encoding	Equivalent	DIVERGE	No	False positive; EBCDIC normalization was incomplete.

TABLE III
DIAGNOSTIC METRICS FOR V1 (MISMATCH IS POSITIVE).

Measure	Value
Accuracy	$8/10 = 0.80$
Mismatch precision	$3/(3+1) = 0.75$
Mismatch recall	$3/(3+1) = 0.75$
Mismatch F1	0.75
Specificity	$5/(5+1) = 0.83$

VSAM, JCL, load-module linkage, and scheduler behavior remain outside v1.

Comparator validity. Normalizers can produce false positives when too weak and false negatives when too permissive. This is why STRUST hashes the comparator and retains the pre-normalized traces.

Scalability. Per-test container startup, compilation, artifact storage, and state reset can dominate runtime for small programs. The slice planner and staged execution fabric are a proposed architecture, not a demonstrated throughput result. Version 1 does not establish graph-extraction accuracy, parallel-scheduling behavior, frontier-closure rate, or the storage cost of long-lived evidence ledgers.

IX. ROADMAP

The next implementation phase should prioritize trustworthiness before breadth. First, build an inventory parser and symbol resolver for copybooks, paragraphs, static and dynamic calls, JCL steps, datasets, and compiler options. Second, construct the system dependence graph and import observed call/data edges from reference traces, retaining unresolved edges as first-class frontier items. Third, implement schema-aware adapters for EBCDIC, zoned decimal, COMP-3, fixed-length records, relational state, and selected VSAM-like abstractions. Fourth, add deterministic fixture synthesis, staged state reset, and context-keyed paired execution so slices can be recomposed without losing provenance. Fifth, integrate the contract and verdict API into continuous integration, where DIVERGE and INCONCLUSIVE outcomes can enforce different release policies. Finally, evaluate on a blinded, larger corpus with published fixtures, seeded faults, and a preregistered comparison policy.

X. CONCLUSION

STRUST-v1 reframes COBOL-to-Java verification as a disciplined differential-evidence problem. Its central idea is simple but consequential: a comparison is credible only when the workload, runtime, representation policy, and observation scope are explicit. For large systems, the unit of trust cannot be a source file alone. It must be a context-aware verification slice that binds a business observation to the code, data, call path, environmental boundary, and byte/field-level evidence that determine it. The version-1 demonstrator validates the mechanics of paired execution, trace capture, normalization, comparison, and reporting; it does not yet validate the proposed scale-out system. Its 80% result is intentionally reported with its two defects, because an honest verifier must make its blind spots visible. With a dependence-graph planner, boundary adapters, staged recomposition, and production environment contracts, STRUST can become a practical quality gate for migrations that need more than a successful build to justify trust.

REFERENCES

[1] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.
[2] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.
[3] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 4, pp. 367–375, 1985.
[4] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.
[5] IBM, "Controlling how numeric data is stored," *Enterprise COBOL for z/OS Documentation*, 2024.
[6] GnuCOBOL Project, "GnuCOBOL Programmer's Guide," ver. 3.2, 2022.
[7] Oracle, "Class BigDecimal (Java SE 11 & JDK 11)," 2024.
[8] Docker, Inc., "Docker documentation," 2024.
[9] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.
[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.
[11] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–60, 1990.
[12] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.