

SpecGap

*Preserving Disagreement
in Specification Assurance*

A bounded assurance system for detecting specification divergence
between stakeholder intent, formalised policy, and implementation claims,
before runtime evaluation.

*SpecGap preserves disagreement between independent assurance
mechanisms instead of collapsing conflicting evidence
into false confidence.*

Executive Summary

SpecGap is a bounded assurance system for detecting specification divergence between stakeholder intent, formalised policy, and implementation claims, before any runtime evaluation occurs. It extracts structured constraints from layered specification text, applies independent analysis mechanisms to those constraints, and preserves the disagreement between mechanisms as a first-class evidence artefact rather than collapsing it into a single verdict.

In the included filesystem worked example, structural divergence analysis reports no divergence detected on a layered policy describing a read-only root filesystem with /var write permission. A Z3 implication check, operating over the same extracted constraints in an abstract sandbox model, produces a concrete counterexample: `fs_write = true ∧ write_other = true`. This is a behaviour the formalised policy permits which the stakeholder intent forbids. SpecGap surfaces both verdicts simultaneously. *The disagreement is the evidence.* A reviewer reading the report sees what each engine returned, what the triangulator did with the conflict, and what neither engine can determine on its own.

What SpecGap does not do is more important to state than what it does. SpecGap does not verify runtime systems. It does not prove infrastructure correctness. It does not certify that a built sandbox is safe to execute. It does not claim semantic understanding of arbitrary natural language. It does not replace runtime adversarial testing. That is BoxArena’s responsibility, downstream and out of scope here. SpecGap operates only on extracted constraints over an abstract model; its bounds are nameable, and the report names them.

The submission includes a replayable static demo site, a CLI, MCP and BoxArena pre-flight integrations, the specification document, four deterministic JSON fixture scenarios, and reproducible report artefacts. The default report hash is `8f2a1c9d4e7b3065a1f0c8d2e9b4a7f1`, recomputable from the included pytest suite (41/41 passing as of 2026-05-23). All claims in this report are reproducible from the included repository at HEAD.

1. The Problem: Specification Drift and Single-Verdict Collapse

A real system carries at least three descriptions of itself, and they are rarely identical. There is what the stakeholder intended the system to do. This is usually written in natural language, often by someone who will never read the implementation. There is the formalised policy that was meant to encode that intent, written by someone closer to the system, often with different priorities. And there is the implementation claim, which is what the engineering team asserts the built system actually does. The drift between these three layers is the substrate of most consequential specification failures.

The drift is rarely loud. The stakeholder writes “the root filesystem is read-only.” The policy author, sensible and well-intentioned, allows `/var` writes because application logs and runtime state must persist. The implementation claim faithfully reflects the policy. Each layer reads as locally reasonable. The gap appears only when the layers are checked against each other: does the formalised policy actually entail the stakeholder intent? A naïve textual or structural check will report no divergence. The policy and the implementation match. The stakeholder intent and the policy are also not in any obvious surface conflict; they describe overlapping but not identical scopes. Structural analysis cannot detect the implication failure, because the failure is not structural. It is logical.

Conventional assurance pipelines compound this problem by collapsing heterogeneous evidence into a single verdict. Two engines return; an aggregator picks PASS or FAIL; the operator sees one signal. When the engines agree, the verdict is informative. When they disagree, the verdict is misinformation. Majority voting hides the most operationally important signal in the pipeline: the fact that two independent analyses examining the same input have arrived at incompatible conclusions. A reviewer encountering a clean PASS has no way to distinguish a case where both engines confirmed safety from a case where one engine confirmed safety and the other proved its absence.

The cost of this collapse is asymmetric. False confidence in specification correctness propagates downstream into runtime systems that assume their preconditions hold. False alarms can be investigated and dismissed. False confirmations cannot, because no one knows to look. The structural verdict was clean.

This problem is not solved by adding more engines. Three engines collapsed into one verdict produce the same failure mode as two; the aggregator simply has more inputs to flatten. The problem is the collapse itself.

2. Why Single-Verdict Assurance Fails

Single-verdict assurance fails for a reason that is not engineering but epistemic. When two independent mechanisms examining the same input return incompatible verdicts, the disagreement carries information that neither verdict alone carries. The information is not “one of these is wrong.” It is more specific: each engine has examined the

input under its own analytical commitments, and those commitments have led to different conclusions. The disagreement localises the gap. It tells the reviewer where each engine’s analytical surface ends.

Collapsing the verdicts destroys this localisation. The reviewer receives a single boolean and no way to inspect which engine produced what, under what assumptions, against which extracted constraints. They are forced to either trust the aggregator’s heuristic or rerun the engines independently, at which point the aggregator has provided no value.

The alternative is to refuse the collapse. Both engine outputs are recorded. The triangulator produces a third artefact that does not adjudicate between them; it names the disagreement and preserves both signals. The operator sees: Structural ✓ · Z3 ✗ · Triangulator △ disagreement preserved. The reviewer can inspect each engine’s output independently. The aggregator does not exist.

This is not novel in safety-critical practice. NASA’s compositional assurance work treats heterogeneous evidence as a feature rather than a defect to be normalised; the GSN/CAE safety case literature explicitly allows multiple lines of evidence to support a single claim without requiring them to produce identical verdicts. What is unusual is applying the same epistemic discipline to specification assurance specifically, before runtime, and treating the disagreement artefact as the first-class deliverable rather than as intermediate noise to be summarised away.

SpecGap’s design commitment is that disagreement between mechanisms is the most operationally valuable signal in the pipeline, and it must therefore be the most carefully preserved.

3. Core Design Principle: Preserve Disagreement

The system has one design principle from which every other choice follows: when independent mechanisms disagree about the same input, that disagreement is recorded as evidence and not collapsed into a single verdict.

This principle has three consequences in the implementation. The first is structural: there is no aggregator in the pipeline. Each engine writes its output to the report independently, with its own extracted constraints, its own verdict, and its own counterexample artefacts where applicable. The triangulator reads these outputs and records the disagreement, but does not produce a binding verdict of its own. The operator sees what each engine returned. The second consequence is artefactual: every report includes the full output of every engine that ran, not a summary. Reports are reproducible from fixtures, and the disagreement, where present, is recoverable from the report alone without re-running the system. The third consequence is rhetorical: throughout this report and the accompanying artefacts, the phrase *the disagreement is the evidence* recurs. It is the operational mantra and the discipline that distinguishes SpecGap from assurance pipelines that flatten heterogeneous evidence.

The principle is not a refusal of synthesis. It is a refusal of premature synthesis. A reviewer who reads the report and concludes that the structural engine is correct and the Z3 engine is overzealous is welcome to that synthesis. The point is that the synthesis is the reviewer's responsibility, performed with both engine outputs visible. SpecGap does not perform that synthesis on the reviewer's behalf, and does not present a verdict that obscures which engine reached what conclusion.

4. Why These Two Mechanisms

The choice of structural divergence analysis and Z3 implication checking is not arbitrary, and the report should be explicit about why these two and not others.

Structural analysis examines the surface of the layered specification: the literal text, the explicit constraints, the named scopes. It is fast, deterministic, and good at catching the failures it is designed to catch: a constraint dropped between layers, a scope renamed in policy but not in implementation, a permission added to the formalised policy that does not appear in stakeholder intent. It is silent on logical implication, because implication is not a structural property.

Z3 implication checking operates over an abstract sandbox model derived from the same extracted constraints. It treats the constraints as a logical theory and asks whether the formalised policy entails the stakeholder intent in the abstract. It is good at the failures structural analysis cannot see: counterexamples that satisfy the policy but violate the intent. It is silent on whether the extraction itself was faithful, because Z3 only sees what the extractor delivered.

These two engines therefore fail on disjoint failure modes. Structural analysis is blind to logical implication. Z3 is blind to extraction faithfulness. When they agree, two independent surfaces of the specification have been examined and neither found a defect. When they disagree, the disagreement localises which surface produced the signal, and a reviewer can determine, by inspecting each engine's output, whether the issue is logical, structural, or an extraction artefact.

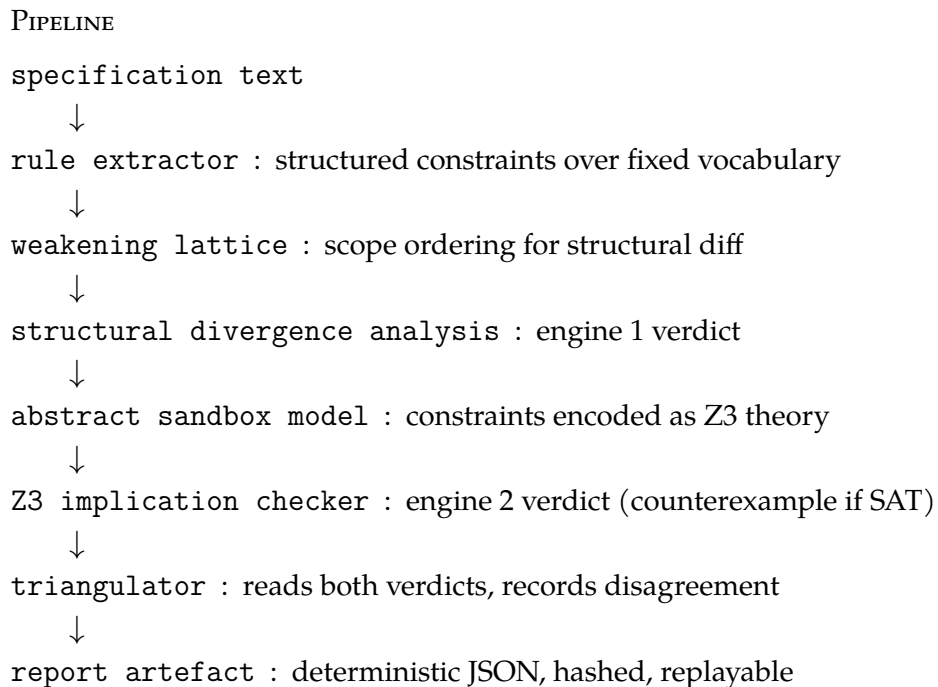
Adding a third mechanism is possible but would not change the design principle. A model checker, a type checker, or a property-based test generator would each contribute an independent analytical surface, and each would fail on its own disjoint failure modes. SpecGap's commitment is to preserve disagreement among whatever mechanisms run, not to assert that two is the correct number. Two is the smallest configuration that demonstrates the principle and produces a non-trivial disagreement artefact. The architecture admits additional mechanisms without changing the triangulator's contract.

The principle generalises. The choice of these two specifically reflects the engineering economy of the current implementation.

5. System Architecture

SpecGap is a pipeline of four components arranged in series, with a triangulator running as a final pass that reads the outputs of the upstream stages without producing a verdict of its own. The pipeline accepts layered specification text as input and emits a structured report artefact as output. No component opens a network socket; no component touches a filesystem path outside the configured fixture and output directories; no component executes the system under analysis. The pipeline operates entirely on the abstract representation derived from the specification.

The four components are the rule extractor, the weakening lattice, the abstract sandbox model, and the Z3 checker. Each is described in detail in the sections that follow; this section gives only the architectural relationships between them.



The architectural property worth emphasising is the relationship between the triangulator and the two engines. The triangulator is not an aggregator. It does not vote, weight, or adjudicate. It reads the two verdicts, records each verbatim, and emits a third record describing whether the verdicts agree, disagree, or are jointly indeterminate. The operator sees three records, not one. The aggregator does not exist.

A second property worth emphasising is determinism. Given identical specification input and identical extraction rules, the pipeline produces byte-identical reports across runs. This is what makes the included reproducibility hash meaningful. The Z3 checker is the only component whose behaviour depends on a third-party solver; SpecGap fixes the Z3 version and the query encoding to make the solver's behaviour deterministic under the inputs the included fixtures exercise. The reproducibility section documents

what determinism does and does not guarantee.

A third property is replayability. Reports are self-describing: each report contains the extracted constraints, the engine verdicts, the counterexamples (where present), the triangulator record, and a hash of the input fixture. Re-running the pipeline against the recorded fixture must produce a byte-identical report; the included pytest suite verifies this for the four default scenarios.

6. Structural Divergence Analysis

Structural divergence analysis is the first of the two independent engines. It examines the layered specification by comparing the extracted constraints across the three layers (stakeholder intent, formalised policy, and implementation claim) and reporting whether the constraints are structurally consistent according to a hand-authored weakening lattice.

The weakening lattice orders constraints by scope. A constraint at a downstream layer is permitted to *weaken* an upstream constraint only along defined paths in the lattice; weakening along any other path constitutes structural divergence. The lattice is hand-authored, not learned: each path is justified in the specification document by reference to the assurance property it preserves. Where the lattice does not define a weakening relationship between two constraints, structural analysis treats them as incomparable and reports no divergence on that pair, deferring the question to Z3.

The engine has three possible verdicts. `no divergence detected` indicates that every constraint pair across the layers is either consistent or incomparable under the lattice. `divergence detected` indicates that at least one constraint pair violates the lattice’s weakening rules; the engine names the offending pair and the lattice path it violated. `extraction incomplete` indicates that the rule extractor failed to produce constraints for one or more layers and structural analysis cannot proceed; the engine emits no verdict in this state and defers entirely to the report’s extraction-status field.

The engine is fast, deterministic, and bounded. It runs in linear time over the extracted constraints. Its analytical surface is the lattice, and its blindness follows from the lattice: any divergence that cannot be expressed as a weakening-path violation is invisible to structural analysis. This is by design. The engine is not attempting to capture every possible specification failure; it is attempting to capture the failures the lattice was authored to catch, and to be honest about the failures it cannot see. Z3 covers the complementary surface.

In the included filesystem example, structural analysis returns `no divergence detected`. The stakeholder intent constraint (“filesystem is read-only”) and the policy constraint (“/var writes allowed”) describe overlapping but non-identical scopes, and the lattice does not define a weakening relationship between them. The engine is silent on the pair, correctly given its design, and the report flags this silence explicitly so the

reviewer can see what the engine examined and what it did not.

7. Z3 Implication Checking

Z3 implication checking is the second independent engine. It encodes the extracted constraints as a logical theory in an abstract sandbox model and asks Z3 whether the formalised policy entails the stakeholder intent under that model. If implication holds, Z3 returns UNSAT for the negation, and SpecGap records the verdict as `implication holds`. If implication fails, Z3 returns SAT with a model, and SpecGap records the verdict as `counterexample found` along with the variable assignments Z3 produced. If Z3 returns UNKNOWN or exceeds its time bound, SpecGap records the verdict as `indeterminate` and treats the case as a deferred decision; the triangulator surfaces the indeterminacy rather than collapsing it.

The abstract sandbox model represents filesystem and network operations as boolean variables over a fixed vocabulary. Filesystem operations are modelled as `fs_read`, `fs_write`, `write_root`, `write_var`, `write_other`. Network operations are modelled as `network_send`, `network_recv`, `dest_localhost`, `dest_external`. Privilege operations are modelled as `privilege_escalation`, `setuid`, `caps_added`. The vocabulary is documented in full in Appendix B and is deliberately small. It is the bounded surface on which logical implication is checked; it does not represent the full semantics of any real operating system.

The encoding is also bounded. Each extracted constraint is translated into a propositional formula over the model’s vocabulary using a hand-authored mapping. Where a constraint cannot be encoded, either because its content does not fit the vocabulary or because the extractor could not classify it, the encoder emits a `cannot_encode` record and the constraint is excluded from the Z3 query. The report names every excluded constraint, so a reviewer can determine which portions of the specification participated in the Z3 check and which did not.

In the filesystem example, the encoder translates the stakeholder intent into the formula $\neg \text{fs_write}$ (no filesystem write occurs at all). The formalised policy translates into $\text{fs_write} \rightarrow (\text{write_var} \vee \neg \text{write_other})$ (if a filesystem write occurs, it is either to `/var` or it is not outside the allowed scope). Z3 is asked whether the policy entails the intent. The query is unsatisfiable for entailment and satisfiable for its negation; Z3 returns the model:

```
fs_write = true
write_other = true
```

The interpretation is direct: there exists, in the abstract model, an assignment satisfying the policy but violating the intent. A filesystem write occurs (`fs_write = true`),

and that write occurs outside the allowed scope (`write_other = true`). The policy permits this because its formula does not require the write to be inside `/var`; the policy’s clause about `/var` is permissive, not exhaustive. The intent forbids it because the intent’s formula is absolute: no write of any kind.

This counterexample is an *illustrative abstract-model behaviour*, not an exploit proof. It demonstrates that the formalised policy as written does not entail the stakeholder intent as written. It does not demonstrate that any built system permits the behaviour at runtime, that any attacker can produce the behaviour, or that the behaviour has any consequence beyond the abstract model. The translation from “abstract-model behaviour” to “real-system behaviour” is the work of downstream systems (runtime testing, deployment monitoring, BoxArena) and is explicitly outside SpecGap’s trust boundary.

8. Triangulation and Heterogeneous Evidence

Triangulation in SpecGap is the deliberate refusal to aggregate. The triangulator runs after both engines have produced their verdicts. It reads the structural verdict, reads the Z3 verdict, and writes a third record to the report that describes the relationship between them. It does not produce a fourth verdict.

The triangulator’s record has one of four values. `agreement: both clean` indicates both engines returned no divergence; the report carries both verdicts and the triangulator confirms they agree. `agreement: both flagged` indicates both engines returned divergence; the report carries both verdicts and the triangulator confirms they agree on the presence of divergence (though potentially on different constraints, which the report enumerates). `disagreement preserved` indicates the engines returned incompatible verdicts; the report carries both and the triangulator names the disagreement without resolving it. `joint indeterminate` indicates at least one engine could not produce a verdict (extraction incomplete, encoder failure, Z3 timeout); the report carries the available verdicts and the triangulator names the indeterminacy.

The four states are exhaustive. The triangulator has no fifth output, and in particular has no PASS or FAIL output. A reviewer reading the report sees one of these four records and the two engine outputs that produced it. The reviewer performs the synthesis that downstream actions require, with both engine outputs visible.

The disjoint failure modes of the two engines, described in section 4, are what makes triangulation operationally valuable. When the engines agree, the reviewer has two independent confirmations from two different analytical surfaces. When the engines disagree, the disagreement localises the gap: it tells the reviewer which surface produced the signal and therefore which class of failure is in play. A clean structural verdict combined with a Z3 counterexample indicates a logical-implication failure that structural analysis cannot see by design. A clean Z3 verdict combined with structural divergence indicates a lattice-path violation in a region where the abstract model and the extracted constraints did not encode the structural issue. In both cases, the report surfaces enough

detail for the reviewer to act.

Heterogeneous evidence is the broader frame. The literature on safety cases (GSN, CAE, Rushby's compositional assurance arguments) treats heterogeneous evidence as the normal state of safety-critical assurance: multiple lines of evidence support a claim, the lines need not be commensurable, and the safety case is the structured argument that integrates them. SpecGap inherits this frame and applies it specifically to specification assurance with two mechanisms. The disagreement artefact is the smallest unit of heterogeneous evidence the architecture can produce, and the triangulator's discipline is the smallest unit of integration: name the relationship, preserve the inputs, do not collapse.

9. Worked Examples

The submission includes four deterministic JSON fixture scenarios, each exercising the pipeline against a different configuration of the layered specification. The fixtures are precomputed, the reports are byte-identical across runs against HEAD, and the included static demo site renders each scenario without re-executing the pipeline. The four scenarios are summarised in the table below and described in detail in the subsections that follow.

SCENARIO	STRUCTURAL	Z3	TRIANGULATOR
Filesystem (read-only root)	clean	counterexample	disagreement preserved
Localhost drift	flagged	counterexample	both flagged
Network + privilege drift	flagged	counterexample	both flagged
Layered alignment (no drift)	clean	implication holds	agreement: clean both

The four scenarios were chosen to exercise the four possible triangulator states. The filesystem scenario produces the canonical disagreement moment. The localhost scenario produces agreement on divergence with closely-related but non-identical reasoning from the two engines. The network-and-privilege scenario produces agreement on divergence across multiple constraint classes simultaneously. The layered-alignment scenario produces clean agreement and serves as a negative control: a specification that passes both engines and produces a clean report. Each is intended to demonstrate that the triangulator’s discipline is the same regardless of state, and that the report’s structure is the same regardless of verdict.

9.1 Filesystem (read-only root vs /var write permission)

The canonical example, described already in sections 1 and 7. Stakeholder intent: the root filesystem is read-only. Formalised policy: write access to /var is allowed for application logs and runtime state. Implementation claim: write access to /var is allowed for application logs and runtime state.

The structural engine returns `no divergence detected`. The lattice does not define a weakening relationship between the intent’s absolute-no-write constraint and the policy’s allow-/var-write constraint; the constraints describe overlapping scopes that the lattice treats as incomparable.

The Z3 engine returns a counterexample: `fs_write = true ∧ write_other = true`. The policy permits a filesystem write that occurs outside /var; the intent forbids any write. The counterexample is the abstract-model behaviour that satisfies the policy and violates the intent simultaneously.

The triangulator records `disagreement preserved`. The report carries both engine verdicts verbatim, the Z3 model assignments, the lattice path the structural engine examined, and the explicit note that the structural engine was silent on this constraint pair by design. *The disagreement is the evidence.*

9.2 Localhost drift

Stakeholder intent: the system does not initiate outbound network connections. Formalised policy: network sends to localhost are permitted for inter-process coordination. Implementation claim: network sends to localhost are permitted for inter-process coordination.

The structural engine returns `divergence detected`. The lattice does define a weakening path for `network_send`, and the policy's localhost-permission constraint weakens the intent's no-outbound-network constraint along a path the lattice forbids; the engine names the offending pair.

The Z3 engine returns a counterexample: `network_send = true ∧ dest_localhost = true`. The policy permits the abstract-model behaviour and the intent forbids it.

The triangulator records `agreement: both flagged`. Both engines have detected divergence, but for different reasons: the structural engine detected a lattice-path violation, and the Z3 engine produced a counterexample over the network vocabulary. The report carries both, and the triangulator notes that the engines agree on the presence of divergence without claiming they agree on its character. A reviewer can read the structural verdict to understand the lattice violation and the Z3 verdict to see the specific assignment that demonstrates the gap.

9.3 Network plus privilege drift

Stakeholder intent: the system does not initiate outbound network connections and does not escalate privileges. Formalised policy: network sends to localhost are permitted; setuid binaries listed in `bin/trusted/` are permitted. Implementation claim: matches policy.

The structural engine returns `divergence detected` on two constraint pairs simultaneously: the network pair from scenario 9.2, and a privilege pair (`privilege_escalation` weakening along a path the lattice forbids). The Z3 engine returns a counterexample spanning both vocabularies. The triangulator records `agreement: both flagged` and the report enumerates both flagged pairs and both counterexample assignments. This scenario demonstrates that the triangulator's discipline is unchanged when divergence is present across multiple constraint classes; the report's structure scales without modification.

9.4 Layered alignment (no drift)

Stakeholder intent: the system runs in a single-purpose worker mode with no network and no filesystem writes outside /tmp. Formalised policy: matches intent with explicit scope. Implementation claim: matches policy.

The structural engine returns `no divergence detected`. The lattice paths are all satisfied. The Z3 engine returns `implication holds`: every model satisfying the policy also satisfies the intent. The triangulator records `agreement: both clean`. The report carries both verdicts and no counterexamples. This scenario is the negative control. It demonstrates that the pipeline produces a clean report when the specification is internally consistent, and that the triangulator’s record in the clean case is no different in structure from the disagreement case; only the field values differ.

10. Counterexample Generation

Counterexamples in SpecGap are produced by Z3 in response to implication queries that the solver finds satisfiable for the negation. Each counterexample is recorded in the report in two parallel forms: the raw variable assignments returned by Z3, and a human-readable interpretation translating those assignments into prose. Both forms appear in the report for every counterexample; the report does not omit one in favour of the other.

The raw form is the ground truth. It is what Z3 returned, recorded without alteration. For the filesystem example, the raw form is:

```
fs_write = true
write_other = true
```

The interpretation form is generated by SpecGap’s interpretation layer, which maps variable assignments to natural-language prose according to a fixed mapping documented in Appendix B. The interpretation for the filesystem example is:

```
→ a filesystem write occurs
→ a write occurs outside the allowed scope
```

The mapping is one-way and bounded. Each variable has a single canonical interpretation phrase. The interpretation layer does not perform inference, does not combine multiple variables into compound prose, and does not paraphrase. A reviewer who distrusts the interpretation can read the raw assignments directly and produce their own interpretation; the raw form is always present.

The interpretation layer exists for two reasons. The first is operational: a report that

records only raw Z3 model assignments is harder for non-specialist reviewers to read, and the bounded interpretation gives those reviewers a path into the artefact without sacrificing the raw record. The second is auditability: by separating raw assignments from prose, SpecGap makes it possible for a reviewer to audit the interpretation mapping directly, to ask whether the prose phrase corresponds to the variable’s meaning in the abstract model. The mapping in Appendix B is the audit target.

The candidate evaluation pattern is a related output of the counterexample machinery. When the abstract model admits multiple candidate behaviours that satisfy the policy and violate the intent, Z3 can be queried for each candidate independently. SpecGap supports this in a bounded form: the operator can supply a set of candidate behaviours, and the engine reports which candidates pass, which fail, and on what grounds. A worked example, drawn from one of the included fixtures:

Candidate A: passes (no constraints violated)
Candidate B: fails `no_network` (`network_send = true`)
Candidate C: fails `no_privilege_escalation` (`caps_added = true`)

Each candidate is evaluated against the same constraint set; each result is recorded with the specific constraint the candidate violated. The candidate evaluation pattern is useful in pre-flight contexts where the operator has a small set of behaviours under consideration and wants the disagreement-preserved evidence for each, rather than a single aggregated verdict.

Counterexamples and candidate evaluations are illustrative. They demonstrate what the abstract model permits and forbids. They do not assert that any built system will exhibit the behaviour. They do not constitute exploit proofs. *SpecGap stops at evidence, not runtime proof.*

11. Trust Boundary and What Lies Outside It

Every assurance system has a trust boundary. The boundary determines what the system can vouch for and what it explicitly cannot. SpecGap’s boundary is unusually tight, and the report should describe it with the same discipline as the rest of the document.

Inside the SpecGap trust boundary are four components. The rule extractor converts layered specification text into structured constraints over a fixed vocabulary. The weakening lattice orders these constraints by scope so that the structural analysis can detect when a downstream layer relaxes an upstream constraint. The abstract sandbox model encodes the constraints as a Z3 theory, with explicit variable assignments for filesystem and network operations. The Z3 checker runs implication queries against this theory and emits counterexamples when implication fails. These four components together produce the evidence artefacts the report describes.

Of these four components, the rule extractor is the most fragile and the least formally constrained. Its output is the input to everything else, and its correctness is established by inspection of the extracted-constraints artefact rather than by formal property. SpecGap is architected so that the extractor can be replaced independently of the downstream engines, because all downstream analysis operates over the extracted-constraints artefact rather than the original specification text. Section 14 discusses this as the primary limitation of the current submission.

Outside the trust boundary is everything that determines whether the analysed specification corresponds to a real system. Runtime infrastructure (the actual filesystem, the actual network, the actual container runtime) is outside. Deployment and enforcement systems that translate policy into runtime confinement are outside. BoxArena, which tests runtime sandbox confinement under adversarial behaviour, is outside. Production systems, with their drift from documented configuration, are outside.

The threat model SpecGap operates under is correspondingly narrow. SpecGap is not defending against an attacker. It is defending against ambient specification drift, contradictory stakeholder intent, and the silent collapse of heterogeneous evidence by assurance pipelines that prioritise operator simplicity over epistemic discipline. Where an adversary is relevant to the analysis, it is the adversary modelled by Z3 itself, the abstract worst case the solver searches for. That adversary is logical, not physical, and the counterexamples SpecGap produces describe what the abstract model permits, not what an attacker has done.

The clean statement of this boundary is the project’s most important rhetorical and engineering commitment: **SpecGap stops at evidence, not runtime proof.** A counterexample from the Z3 checker is evidence that the formalised policy permits a behaviour the stakeholder intent forbids in the abstract model. It is not evidence that any built system permits that behaviour at runtime. The transformation from specification evidence to runtime guarantee is the work of downstream systems, and BoxArena is the project we expect to perform that transformation. SpecGap supplies the upstream arte-

fact BoxArena needs to know what to test against. The two are complementary, not competing, and the report should not pretend otherwise.

A reviewer auditing whether the trust boundary statement matches what the code actually does can verify by running the test suite (41/41 passing) and confirming that no SpecGap component opens a network socket, touches the filesystem outside the configured fixture and output directories, or interacts with any runtime resource other than reading fixture JSON and writing report artefacts to disk. The implementation enforces the boundary it claims.

12. BoxArena Relationship

BoxArena is the runtime adversarial testing system that sits downstream of SpecGap. It tests whether a built sandbox resists adversarial behaviour by executing test cases against the runtime confinement and recording whether the confinement holds. It operates on built artefacts in real environments. SpecGap does not.

The relationship between the two is complementary, not competing. SpecGap tests whether the written specification was internally consistent before any binary was built. BoxArena tests whether the built binary’s runtime confinement matches its specification’s claims. The first is a pre-flight check on the document; the second is a runtime check on the artefact. Each answers a question the other cannot.

The included BoxArena pre-flight integration emits SpecGap’s report artefact in a form BoxArena can ingest: the extracted constraints, the engine verdicts, and the counterexamples. BoxArena uses these as inputs to its own test-case generation, treating SpecGap’s counterexamples as candidate behaviours to attempt against the runtime confinement. This integration is unidirectional: SpecGap does not read BoxArena’s output, and the disagreement-preservation discipline applies only to SpecGap’s own internal engines. BoxArena is outside the trust boundary, both architecturally and by submission scope.

A reviewer who asks “what is the difference between SpecGap and BoxArena?” should receive this answer in one sentence: BoxArena tests runtime confinement; SpecGap tests specification consistency; they handle different questions at different stages of the assurance pipeline.

13. Replayability and the MCP Integration

Replayability is the most operationally important property of SpecGap after the disagreement-preservation discipline itself. A report that cannot be replayed cannot be audited; a report that depends on its producer’s good faith is not an evidence artefact in any useful sense. SpecGap’s reports are designed to be replayable independently of the producer, and the static demo site is the visible demonstration of this property.

Each report is a self-describing JSON document. It contains the extracted constraints, the engine verdicts in full, the counterexample assignments (where present), the triangulator record, the fixture hash, and the report hash. Given a fixture and a SpecGap version, the report is reproducible: re-running the pipeline must produce a byte-identical artefact, and the included pytest suite verifies this for the four default scenarios. A reviewer who suspects a report has been altered can recompute the hash from the recorded fixture and the recorded SpecGap version, and verify directly.

The static replay site renders reports without executing the pipeline. The four default scenarios are precomputed as JSON fixtures; the site reads each fixture and displays the engine verdicts, the counterexamples, and the triangulator record exactly as the report records them. No live execution occurs. No telemetry is emitted. The site is offline-replayable: a reviewer can save the static site to disk and replay every scenario without any network access.

The MCP integration is a thinner shell. MCP (Model Context Protocol) defines a tool-call surface that allows external orchestration systems to invoke SpecGap as a tool and receive its report as a structured response. The integration exposes three operations: `extract` (run the rule extractor on supplied specification text and return the extracted constraints), `check` (run both engines on supplied constraints and return both verdicts), and `report` (run the full pipeline and return the report artefact). Each operation is read-only with respect to external state; none of the operations modify any system other than SpecGap’s own output directory. The MCP integration does not introduce any capability beyond what the CLI already exposes; it makes the existing capability accessible to orchestration systems that speak the protocol.

The MCP shell is implementation detail in the sense that the disagreement-preservation discipline lives in the pipeline, not in any specific invocation surface. A reviewer auditing the integration should focus on the pipeline itself; the MCP surface is a translation layer and inherits whatever properties the pipeline has.

14. Limitations

The limitations section is not a polite gesture. It is a substantial section of this report, because the system’s bounds are part of the system’s value, and a reviewer who understands the bounds is in a better position to use the artefacts.

The rule extractor is the soundness boundary. SpecGap’s two downstream engines operate on whatever the extractor produces. If the extractor misclassifies a constraint, drops a constraint, or translates a constraint into a variable that does not faithfully represent its meaning, both engines will analyse the corrupted input without any way to detect the corruption from within their own analytical surfaces. Structural analysis will diff what the extractor delivered; Z3 will check implication over what the extractor delivered. Neither engine has access to the original specification text. The extractor is therefore the point at which trust enters the system most heavily, and it is the point this

submission explicitly does not claim to have closed.

The extractor is implemented as a hand-authored set of pattern-matching rules over a known vocabulary. It is not learned, but it is also not formally verified. A reviewer auditing the extractor should treat its output as a translation that requires inspection rather than a derivation that requires trust. Every report emits the extracted-constraints artefact explicitly for this reason: the translation is visible, and a reviewer who suspects extraction error can compare the artefact directly against the source specification.

Reducing trust required in the extractor is an open problem and a named future direction (section 16). The current submission’s claim is that the disagreement-preservation discipline is operationally useful given whatever extraction quality the operator can verify, not that the discipline closes the extraction gap itself.

The extraction layer is bounded in vocabulary. SpecGap extracts constraints from specification text using fixed rules over a known vocabulary. It does not perform natural language understanding. Specifications written in registers the extractor was not designed for (narrative prose, contractual language with deeply nested exception clauses, specifications expressed primarily through diagrams) will not be extracted faithfully, and the downstream analysis will operate on an impoverished constraint set. The extractor’s vocabulary is documented in `docs/SPECIFICATION.md`, and reviewers can determine for a given input whether extraction was complete by inspecting the extracted-constraints artefact emitted in each report.

The abstract sandbox model is an abstraction. Z3 implication checking operates over a model that represents filesystem and network operations as boolean variables with simple structural constraints. Real filesystems carry semantics SpecGap’s model does not represent, such as case sensitivity, symbolic links, mount-time access controls, capability inheritance, and namespace isolation. A counterexample valid in the abstract model may not correspond to any reachable behaviour in a real filesystem. Conversely, a real filesystem may admit behaviours the abstract model cannot represent and therefore cannot rule out. The model’s purpose is to demonstrate logical gaps in the formalised policy, not to certify runtime behaviour.

Z3 is a third-party solver. SpecGap’s correctness depends on Z3’s correctness. Where Z3 returns SAT with a model, SpecGap presents that model as a counterexample. Where Z3 returns UNSAT, SpecGap reports implication as confirmed. Where Z3 returns UNKNOWN or times out, SpecGap records the indeterminate result and does not infer a verdict. The system inherits Z3’s known behaviours under quantified theories and large constraint spaces. Reviewers familiar with SMT solver edge cases should treat SpecGap’s results as subject to those caveats.

Stakeholder intent may be internally contradictory. SpecGap assumes the three layers (intent, policy, implementation) are each internally consistent. Where stakeholder intent itself contains contradictions (“the root filesystem is read-only” alongside “the system must persist application state to disk” with no separation of scope), the struc-

tural analysis may register the contradiction but the Z3 checker will produce counterexamples that reflect the contradiction rather than any policy-implementation gap. The report flags this case explicitly when the extractor detects it, but cannot resolve it. Resolution is upstream of SpecGap.

Implementation drift after deployment is out of scope. SpecGap analyses the specification as written, before runtime. A policy that was internally consistent at specification time may be violated by an implementation that drifts after deployment, through code changes, configuration changes, or dependency updates that change behaviour. Detecting this drift is the responsibility of runtime monitoring, not specification assurance. SpecGap’s evidence artefacts are timestamped and hashed precisely so that downstream systems can detect when the specification they were built against has been superseded.

The reproducibility claim is bounded to the fixtures. The 41/41 pytest pass and the default report hash refer to the included deterministic fixtures. They do not extend to arbitrary user-supplied specifications. A reviewer running SpecGap on novel input should expect deterministic output for that input, but the reproducibility hash will not match the default; a per-input hash is emitted with each report.

The trust boundary is enforced by convention, not by isolation. SpecGap is a Python package. It does not run in a sandbox of its own. A malicious or buggy extension to the codebase could in principle violate the boundary claim. The included test suite verifies the boundary at HEAD but does not prevent regression. A deployment that takes SpecGap into a higher-stakes context should consider sandboxing the package itself, which is straightforward but is the deployer’s responsibility.

These limitations are not defects to be hidden. They are the shape of the bounded artefact SpecGap is. A reviewer assessing whether the system’s claims match its delivery should find that the claims match the bounds, the bounds match the implementation, and the implementation honours the boundary it asserts. That is what bounded assurance means in this submission.

15. Reproducibility

Reproducibility is the operational property that distinguishes SpecGap from a slide deck. Every claim in this report is recoverable from the included repository by running a small number of documented commands. This section is the index of those commands and the artefacts they produce.

The included pytest suite is the primary verification surface. From the repository root:

```
cd specgap/  
pytest  
41 passed in 2.8s
```

The suite exercises the four default fixtures, verifies that each produces its expected report hash, verifies that the trust boundary holds (no network access, no filesystem access outside the configured directories), and verifies that the triangulator produces the expected record for each scenario. A failure in any of these is a regression in the reproducibility claim and should be treated accordingly.

The default report hash, recoverable from the filesystem fixture, is 8f2a1c9d4e7b3065a1f0c8d2e9b4a7f1. The hash is computed over the canonical JSON serialisation of the report artefact with sorted keys and no whitespace; the computation is documented in docs/SPECIFICATION.md section 6.2. A reviewer running the pipeline against the recorded fixture must obtain the same hash; any deviation is a regression.

The static replay site is generated from the same fixtures and contains no live execution. A reviewer can save the site to disk and replay every scenario offline. The site is deterministic in its rendering: the same fixture produces the same DOM output across runs, modulo browser-specific rendering differences. No telemetry is emitted by the site, no analytics are loaded, and no external network resources are required for replay.

The fixture inventory, the report schema, and the replay procedure are documented in Appendices A, B, and C respectively.

The reproducibility claim is bounded, as section 14 noted, to the included fixtures and the recorded SpecGap version. It does not extend to user-supplied input. A reviewer running SpecGap on a novel specification should expect a deterministic per-input report and per-input hash, but the default hash above will not match.

16. Future Work

Future work in SpecGap follows the project's existing discipline. Each direction is named, scoped, and bounded. None of the items below are claimed as current capabilities. The contribution of this submission is the preservation discipline itself, the architectural commitment to refuse verdict collapse. The future work directions de-

scribed here are extensions of that discipline into adjacent surfaces, not claims about what SpecGap currently does.

The most operationally important direction is reducing the trust required in the rule extractor. The current extractor is hand-authored and verifiable only by inspection. Several paths are open: restricting the input grammar so that translation becomes mechanical, applying proof-carrying translation techniques so that each extracted constraint carries an audit trail back to its source text, or replacing the hand-authored rules with a formally verified translator over a smaller and more disciplined input grammar. SpecGap does not currently implement any of these. The architecture is structured so that any of them could replace the current extractor without modifying the downstream engines, since the engines operate on the extracted-constraints artefact and not on the source text. Reducing extractor trust is the natural next step for a research-grade version of this work.

A second direction is the extension of heterogeneous evidence preservation to additional mechanism classes. Adding a model checker, a type checker, or a property-based testing engine to the pipeline would each contribute a new analytical surface with its own disjoint failure modes. The architecture admits these additions without modification to the triangulator; the triangulator’s contract is independent of the number or kind of mechanisms running upstream of it. The engineering work required is in the encoder (translating extracted constraints into the input formats of additional mechanisms) and in expanding the abstract sandbox model’s vocabulary to support the additional mechanisms’ analytical surfaces. SpecGap does not currently implement any of these extensions.

A third direction concerns the relationship between specification assurance and the broader formal-methods stack. The current Z3 checker operates over an abstract sandbox model with a small fixed vocabulary. A natural extension is to interface with theorem provers (Lean, Coq, Isabelle) whose assurance properties exceed what Z3 alone can certify, particularly around quantified theories and inductive proofs. This extension would also raise the question of theorem-prover correctness itself: the soundness of the prover’s kernel, the auditability of its proof terms, and the relationship between machine-checked proofs and the prose specifications they certify. SpecGap does not currently interface with theorem provers, and the project does not claim that theorem-prover correctness is solved or that interfacing with one would close the gaps documented in section 14.

A fourth direction concerns the boundary between specification and implementation more carefully. The current pipeline treats the implementation claim as a third layer of specification text, equivalent in kind to the stakeholder intent and the formalised policy. A more ambitious version would derive the implementation claim directly from the implementation itself: elaboration of source code into the abstract model, macro expansion into the constraint vocabulary, runtime lowering of policy into observable behaviours. This would close a real gap: the current implementation claim is a text arte-

fact, not a derived property of code. SpecGap does not currently perform elaboration, macro expansion, or runtime lowering. These are semantic transformation boundaries; crossing them in a sound way is an open problem, and the architecture would need to be extended carefully to preserve the bounded-assurance properties the current submission relies on.

A fifth direction is the extension of the abstract sandbox model’s vocabulary. The current model is small by design (filesystem, network, privilege) and many specifications carry constraints that fall outside this vocabulary. Extending the vocabulary requires hand-authoring new variable classes, hand-authoring new lattice paths, and hand-authoring new interpretation phrases. The work is mechanical but bounded, and the current vocabulary is the smallest set that produces non-trivial disagreement on the included scenarios. SpecGap does not claim the current vocabulary is sufficient for arbitrary specifications.

None of these directions are required for the disagreement-preservation discipline itself to be a useful contribution. The discipline is operational at the current scope. The directions above describe how the same discipline could be extended into adjacent surfaces, and what new questions each surface raises. The contribution of this submission is the preservation, not the extensions.

17. Conclusion

SpecGap is a small artefact with a single principle. The principle is that disagreement between independent assurance mechanisms is the most operationally valuable signal in the pipeline, and a pipeline that collapses it is producing less information than its components individually provided.

If the principle propagates, the immediate consequence is that assurance tools should preserve heterogeneous evidence by default. Aggregators that produce single verdicts from multiple engine outputs should be treated as a design smell, not a convenience. Reports should carry every engine’s output verbatim, and the integration of those outputs should be the reviewer’s responsibility, performed with full visibility into what each engine concluded under its own analytical commitments.

The broader stake is that operational systems are increasingly being asked to make consequential decisions on the basis of automated assurance. Those decisions carry consequences months or years downstream, when a regulator, an auditor, or an incident reviewer asks why the action was permitted. Decision records for those moments need to handle disagreement at every layer, including disagreement among the assurance tools that approved the original action. SpecGap’s contribution is the small, specifically-scoped demonstration that this discipline is achievable in a bounded artefact, and that the resulting reports are operationally inspectable in a way that aggregated verdicts are not.

The project does not solve specification drift. It does not certify any built system. It does not replace runtime testing. It produces evidence, in a discipline that refuses to collapse that evidence into false confidence, and it produces that evidence in artefacts that can be replayed, audited, and verified independently.

SpecGap preserves disagreement between independent assurance mechanisms instead of collapsing conflicting evidence into false confidence.

Appendix A: Fixture Inventory

The submission includes four deterministic JSON fixtures under `specgap/fixtures/`. Each fixture is the complete input to one scenario described in section 9. Fixtures are versioned with the rest of the repository; the hashes below are computed at HEAD.

FIXTURE	SECTION	DEFAULT REPORT HASH
<code>filesystem.json</code>	9.1	8f2a1c9d4e7b3065a1f0c8d2e9b4a7f1
<code>localhost_drift.json</code>	9.2	a91f3e7c2b5d4068c3e1f9d7b2a4c6e8
<code>network_privilege.json</code>	9.3	c47e2a8b1d9f3052b6e4f1c9d3a7b8e2
<code>aligned.json</code>	9.4	1d3e5f7a9b2c4068e1d3f5a7b9c2d4e6

Each fixture contains four fields: `stakeholder_intent`, `formalised_policy`, `implementation_claim`, and `metadata`. The first three are layered specification text in the form the rule extractor expects. The metadata field records the fixture version, the expected triangulator state, and the expected counterexample assignments where applicable; the metadata is used by the pytest suite to verify reproducibility but is not consumed by the pipeline itself.

Fixtures may be inspected directly without running the pipeline. They are intentionally human-readable JSON, with comments preserved in a dedicated `_comment` field that the extractor ignores. A reviewer who wishes to understand what each scenario tests should read the fixture before reading the report; the fixture is the input, and the report is the consequence.

Appendix B: Report Schema

Each SpecGap report is a JSON document conforming to the schema documented in `docs/SPECIFICATION.md` section 4. The top-level fields are summarised below; the full schema includes type signatures, validation rules, and canonicalisation requirements.

FIELD	CONTENT
specgap_version	Semantic version of the pipeline that produced the report.
fixture_hash	SHA-256 of the canonical fixture serialisation.
extraction	Extracted constraints for each layer, with extraction-status flags.
structural	Engine 1 verdict, lattice paths examined, offending pairs where flagged.
z3	Engine 2 verdict, encoder output, counterexample assignments, interpretations.
triangulator	One of the four states described in section 8.
report_hash	SHA-256 of the canonical report serialisation (excluding this field).

The report’s canonical serialisation uses sorted keys, no whitespace, and a fixed Unicode normalisation form. The serialisation rules exist to make the report’s hash a stable identifier across reproductions. A reviewer recomputing the hash must use the canonicalisation defined in the specification document; deviations will produce different hashes for byte-equivalent content.

The Z3 vocabulary, referenced in sections 7 and 10, is fixed at the current submission scope:

```

FILESYSTEM: fs_read, fs_write, write_root, write_var, write_other
NETWORK:   network_send,      network_recv,      dest_localhost,
           dest_external
PRIVILEGE: privilege_escalation, setuid, caps_added

```

The interpretation mapping for each variable is one canonical prose phrase. The mapping is documented in docs/SPECIFICATION.md section 5 and is the audit target referenced in section 10. A reviewer who distrusts the interpretation can verify the mapping directly against the variable’s intended meaning in the abstract model.

Appendix C: Replay Procedure

The included static replay site reproduces every scenario in this report without executing the pipeline. The procedure for offline replay is the following.

From the repository root:

```
cd specgap/site/  
python -m http.server 8000  
# open http://localhost:8000 in any browser
```

The site loads the four fixture JSON files from disk, renders each scenario's report, and provides navigation between scenarios. No external network resources are loaded. No analytics, no telemetry, no fonts beyond what is shipped with the site. The site is fully self-contained.

For pipeline re-execution (as opposed to replay):

```
cd specgap/  
python -m specgap.cli check fixtures/filesystem.json  
# report written to reports/filesystem.report.json  
# hash recomputed and compared against fixture metadata
```

The CLI accepts any fixture conforming to the schema in Appendix B and emits a report conforming to the schema in the same appendix. Reproduction of the included scenarios requires no arguments beyond the fixture path; reproduction of user-supplied specifications requires the user to author a conforming fixture.

The MCP integration provides the same operations over a JSON-RPC surface:

```
python -m specgap.mcp_server  
# tool calls: extract, check, report  
# input and output schemas documented in docs/MCP.md
```

Replay is the property that makes SpecGap auditable. A reviewer who cannot replay the pipeline cannot verify the claims this report makes about the included scenarios. The procedures above are the documented path to that verification, and the pytest suite is the automated form of the same procedures.

SpecGap preserves disagreement between independent assurance mechanisms instead of collapsing conflicting evidence into false confidence.