

SpecGap Arena: Proactive Obligation Synthesis for Spec Adequacy

A benchmark and remediation framework for hidden obligations in AI-generated code

Team : Obligation Cartographers

Authors : Ayush Raj, Rohit Kale, Raviraj Shelar, Parth Nawkar

May 24, 2026

Abstract

AI systems can now generate code and candidate tests at high speed, but a durable bottleneck remains: determining whether the specification captures the intended outcome. We introduce SpecGap Arena, a benchmark and remediation framework for hidden semantic obligations in AI-generated code. The benchmark is built around obligation mutants: plausible implementations that satisfy public prompts and public tests while omitting a critical behavioral constraint. Each controlled task pairs a public-facing specification with a robust reference, a missing-obligation mutant, an obvious-failure control, a named obligation, and a replayable counterexample. Across the controlled core, obligation mutants consistently survive public tests while failing stronger outcome checks; robust references are preserved; and obvious-failure controls are rejected by public tests. These results show how ordinary testing can create false confidence even when implementations appear plausible. SpecGap Arena provides a reproducible setting for measuring this failure mode and demonstrates a practical preflight workflow for surfacing trust boundaries, protected assets, and negative obligations before generated code is accepted.

1. Introduction

Secure program synthesis is not only a code-generation problem. It is also a specification-adequacy problem. Public tests often encode positive examples: a parser handles a simple string, a permission resolver returns a normal allow, or a user importer fills defaults. Security and correctness failures often live in the negative space:

- untrusted input must not create privilege-bearing fields;
- decoded and normalized paths must remain inside a trusted base directory;
- logout must invalidate all privilege-bearing state;
- deny rules must dominate later allow rules;
- malformed parser inputs must not be accepted as ordinary data.

Research question. Can a pre-implementation obligation workflow identify missing semantic constraints that public tests miss, and can the resulting checks reject plausible implementations without rejecting robust references?

2. Contributions

1. **Seventeen-task obligation benchmark.** A benchmark suite spanning parsers, security wrappers, state machines, resource edges, refactor preservation, temporal behavior, idempotency, information leakage, and environment semantics.
2. **Controlled deterministic core.** A 12-task core where every task has public tests, a robust reference, a plausible obligation mutant, a sanity-control failure, a named obligation, and a replayable counterexample.
3. **False-confidence measurement.** A controlled definition: public tests pass, outcome checks fail, and the robust reference is preserved.
4. **Obligation preflight workflow.** A concrete remediation workflow that extracts trust boundaries, protected assets, negative obligations, acceptance checks, and assurance-card evidence before implementation acceptance.
5. **Replayable assurance artifacts.** Deterministic result files, assurance cards, and a demo trace allow reviewers to inspect the benchmark without relying on live model calls.
6. **Extensible evidence layer.** The framework is designed to incorporate property-based testing, symbolic checks, bounded models, and provider sweeps while keeping the core benchmark reproducible.

3. Benchmark Design Rationale

The benchmark tasks are deliberately small, but they are not arbitrary toy examples. Each task is an obligation microscope: it isolates one semantic constraint that can be absent from public examples while still being central to the intended behavior. This makes failures interpretable. A path wrapper can be wrong because it omits decoded containment; a CSV parser can be wrong because it omits quoted-field semantics; a session machine can be wrong because logout clears visible state but leaves privilege-bearing state alive.

Table 1: How the 17-task suite generalizes beyond the controlled core.

Family	Controlled core tasks	Supplementary tasks	Generalizes to
Parser/serializer	t01_parse_query, t08_csv_parse	–	API parsers, protocol decoders, configuration loaders, and malformed-input handling.
Security wrapper	t02_safe_join, t09_permissions, t12_user_deserialize	error_msg_leak	Authorization, sandboxing, import boundaries, and confidentiality-preserving error handling.
State machine	t03_session_state, t04_rate_limit	timezone_expiry	Login/session flows, rate-limit windows, lifecycle workflows, and temporal validity checks.
Resource/accounting	t05_discount_total, t07_merge_defaults	float_discount, default_mutable	Payments, configuration systems, mutable defaults, and financial precision boundaries.
Refactor preservation	t06_dedupe_events, t11_cache_get	idempotent_payment	Compatibility layers, cache semantics, deduplication, and retry-safe service behavior.
RL/environment	t10_grid_env	–	Agent-environment contracts such as reset, step, reward, and terminal-state semantics.

3.1. Why Small Tasks Underestimate Risk

Each controlled task contains one named missing obligation. This makes the experiment interpretable, but it also makes the measured miss rate conservative. In deployed systems, obligations compose across modules: a parser obligation can affect authorization, authorization can affect state transitions, and state transitions can affect persistence, retries, and external side effects. As the number of interacting components grows, the chance that at least one obligation is omitted grows as well. One simple intuition is:

$$P(\text{at least one obligation miss}) = 1 - \prod_i (1 - p_i),$$

where p_i is the probability that obligation i is omitted. This is not a fitted statistical model; it is the reason the benchmark starts with small controlled probes before scaling to composed workflows.

Stable result. SpecGap Arena contains 17 implemented obligation tasks. The 12-task controlled core has 12 controlled false-confidence cases: every obligation mutant survives public tests and is rejected by an outcome check, while every robust reference is preserved.

4. System Overview

The key design choice is to make the core experiment deterministic. Model behavior is important, but it is stochastic and provider-dependent. SpecGap Arena therefore fixes the task, mutant, robust reference, and outcome obligation first; provider sweeps can then be layered on top without making the core result irreproducible.

Table 2: Suite structure.

Slice	Role	Tasks
Controlled deterministic core	Headline false-confidence measurement with robust references, obligation mutants, obvious failures, named obligations, and replayable counterexamples.	12
Supplementary expansion tasks	Additional obligation classes for coverage breadth: financial precision, temporal expiry, idempotency, information leakage, and mutable-default isolation.	5
Implemented suite	Total benchmark inventory.	17

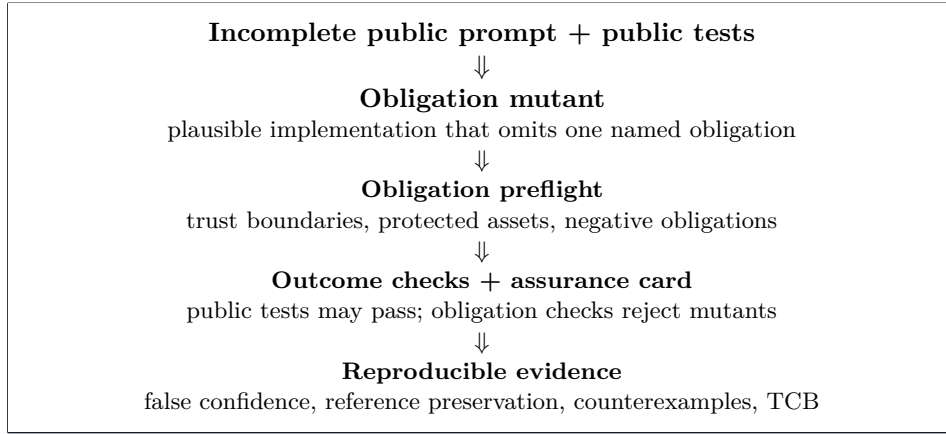


Figure 1: SpecGap Arena pipeline. The deterministic mutant makes the result replayable; optional model and backend evidence can be evaluated against the same fixed obligations.

5. Flagship Demo Task

5.1. Incomplete Public Prompt

Write a function `import_user(record, current_user)` that merges fields from an external record into a new user profile, uses `current_user` as a base template, and returns fields including `name`, `email`, `department`, `is_admin`, and `role`. Default `is_admin` to `False` and `role` to `"user"` if not already set.

5.2. Public Tests

```

assert import_user({"name": "Ada"}, {})[ "is_admin" ] is False
assert import_user({"name": "Ada", "email": "ada@example.com"}, {})[ "role" ] == "user"
assert import_user({"name": "Ada"}, {"department": "research"})[ "department" ] == "research"

```

These tests check normal behavior and defaults. They do not check the authorization boundary.

5.3. Obligation Mutant

```

def import_user(record: dict, current_user: dict) -> dict:
    result = dict(current_user)
    result.update(record)
    if "is_admin" not in result:
        result["is_admin"] = False
    if "role" not in result:
        result["role"] = "user"
    return result

```

This implementation is short, idiomatic, and passes the public tests. It fails the missing obligation:

Missing obligation. External records are untrusted. They must not grant, modify, preserve, or imply privileges through fields such as `is_admin`, `role`, or `permissions`. Unknown fields should be rejected or quarantined, and trusted state must not be mutated.

```
import_user({"name": "Eve", "is_admin": True}, {})
# mutant returns {"name": "Eve", "is_admin": True, "role": "user"}
```

6. Obligation Preflight

Obligation preflight is run before accepting an implementation. It does not assume the public prompt is complete. It extracts a structured obligation model.

Table 3: Preflight output for the flagship task.

Field	Inferred content
Interface	<code>import_user(record, current_user) -> user</code>
Trust boundary	<code>record</code> is untrusted external input; <code>current_user</code> is trusted internal state.
Protected assets	<code>is_admin</code> , <code>role</code> , <code>permissions</code> , authorization invariants.
Negative obligations	Reject privilege injection; reject/quarantine unknown fields; do not mutate trusted state; fail closed on malformed records.
Boundary class	Security-wrapper boundary around an authorization-relevant internal object.

6.1. Executable Obligation Checks

```
assert_raises(ValueError, lambda: import_user({"name": "Eve", "is_admin": True}, {}))
assert_raises(ValueError, lambda: import_user({"name": "Eve", "role": "admin"}, {}))

original = {"name": "Ada", "is_admin": True, "role": "admin"}
before = dict(original)
assert_raises(ValueError, lambda: import_user({"name": "Ada", "role": "user"}, original))
assert original == before
```

6.2. Contract-Style Obligation

The same obligation can be expressed as a lightweight contract wrapper. This is not a separate proof system; it is a structured way to make the preflight obligation executable before accepting an implementation.

```
PRIVILEGE_FIELDS = {"is_admin", "role", "permissions"}

def import_user_contract(record: dict, current_user: dict) -> dict:
    """
    pre: isinstance(record, dict)
    pre: isinstance(current_user, dict)
    post: current_user is not mutated
    post: accepted external records do not carry privilege fields
    """
    before = dict(current_user)
    try:
        result = import_user(record, current_user)
    except ValueError:
        assert current_user == before
        raise

    assert current_user == before
    assert not any(field in record for field in PRIVILEGE_FIELDS)
    assert result.get("role") not in ("admin", "root", "owner")
    assert result.get("is_admin") is not True
    return result
```

In implementation, this can be encoded either as PEP316-style documentation, runtime assertion wrappers, or generated property checks. The important point is that the obligation is made explicit before the implementation is trusted.

6.3. Z3 Bounded Model

Z3 checks an abstract bounded policy model. This is not a Python execution proof; it is an exhaustive check over a deliberately small abstraction. For the flagship task, the abstraction asks whether an untrusted record can contain a privilege-bearing field and still be accepted.

```
from z3 import Bool, Solver, Or, And, Not, sat, unsat

has_is_admin = Bool("has_is_admin")
has_admin_role = Bool("has_admin_role")
has_permissions = Bool("has_permissions")
accepted = Bool("accepted")

privilege_input = Or(has_is_admin, has_admin_role, has_permissions)
policy_violation = And(privilege_input, accepted)

# Mutant policy: acceptance is unconstrained, so a violation exists.
mutant = Solver()
mutant.add(policy_violation)
assert mutant.check() == sat

# Completed policy: privilege-bearing external records are rejected.
completed = Solver()
completed.add(privilege_input)
completed.add(accepted)
completed.add(Not(policy_violation))
assert completed.check() == unsat
```

This bounded model does not replace the executable checks. It complements them by separating the policy obligation from Python implementation details, making it easier to inspect whether the intended rule itself is coherent.

7. Evaluation Design

Each deterministic task has three candidate classes:

Table 4: Candidate implementation classes.

Candidate	Purpose	Expected result
Robust reference	Implements the completed obligation.	Public and outcome checks pass.
Obligation mutant	Omits exactly one named obligation while preserving public examples.	Public tests pass; outcome checks fail.
Obvious failure	Sanity-check implementation that violates public behavior.	Public tests fail.

The controlled-core candidate classes are fixed for replayability. Public prompts, public tests, robust references, obligation mutants, and obvious-failure controls are evaluated against the same outcome obligations, so the public-test gate and the stronger outcome-check gate can be inspected independently. Live model sweeps use the same task obligations but are treated as supporting evidence rather than as part of the deterministic core.

A candidate is counted as a controlled false-confidence case if:

$$\text{PublicTests} = \text{pass} \quad \wedge \quad \text{OutcomeCheck} = \text{fail} \quad \wedge \quad \text{ReferencePreserved} = \text{true}.$$

The controlled core also records review/spec labels. Live model behavior is handled separately by the provider-sweep infrastructure and is not needed to reproduce the core metrics.

8. Verified Results

The 17-task suite demonstrates coverage breadth; the 12-task controlled core supports the headline false-confidence measurement. The controlled core uses two validity gates. Robust references must pass both public and outcome checks, preventing overstrong obligations from rejecting correct implementations. Obvious-failure controls must fail public tests, confirming that public tests still catch basic regressions. Both gates pass on all 12 controlled tasks.

Table 5 summarizes the controlled core. The key comparison is between weak public evidence and stronger outcome obligations: public tests accept every obligation mutant, while outcome checks reject every mutant and preserve every robust reference.

Table 5: Controlled deterministic core metrics.

Metric	Count	Rate
Controlled core tasks	12	–
Candidate-task evaluations	36	–
Public-test passes across all candidates	24/36	66.7%
Outcome-check passes across all candidates	12/36	33.3%
Controlled false-confidence cases	12/36	33.3%
Obligation mutants surviving public tests	12/12	100.0%
Obligation mutants killed by outcome checks	12/12	100.0%
Robust references preserved	12/12	100.0%
Obvious-failure controls caught by public tests	12/12	100.0%
Spec omission among mutant tasks	12/12	100.0%
Spec disagreement among mutant tasks	9/12	75.0%

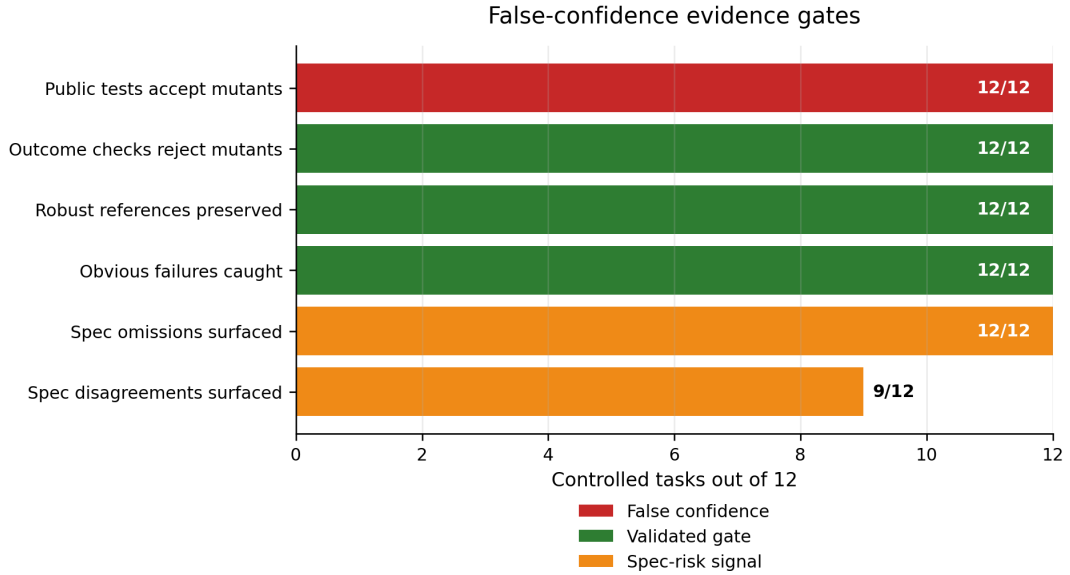


Figure 2: False-confidence evidence gates for the controlled core. Red denotes false confidence, green denotes validation gates that behave as intended, and amber denotes specification-risk signals surfaced before code acceptance.

The specification-level evidence is important because it appears before full implementation proof. All 12 mutant tasks show an obligation-omission signal, and 9 of 12 show disagreement between candidate specifications. The defensible claim is not that models always miss these obligations; it is that ordinary prompts and examples often fail to elicit critical obligations, while structured preflight roles can surface them before code acceptance.

Table 6: Controlled false-confidence cases by task category.

Category	False-confidence cases
Security wrapper	3
Parser/serializer	2
Refactor preservation	2
Resource/edge case	2
State machine	2
RL/game environment	1

8.1. Local Provider-Sweep Evidence

We also include three saved local-provider sweeps over the 17-task suite: `deepseek-coder-v2-lite-instruct`, `qwen/qwen2.5-coder-14b`, and `qwen3.6-35b-a3b`, all through LM Studio at temperature 0. Across 51 model-task evaluations, generated code passed public tests in 40 cases. Of those public-passing implementations, 20 failed obligation checks, giving a 50% public-test false-confidence rate among public-passing generated code. These sweeps are supporting empirical evidence, not the headline controlled result and not a completed cross-provider comparison.

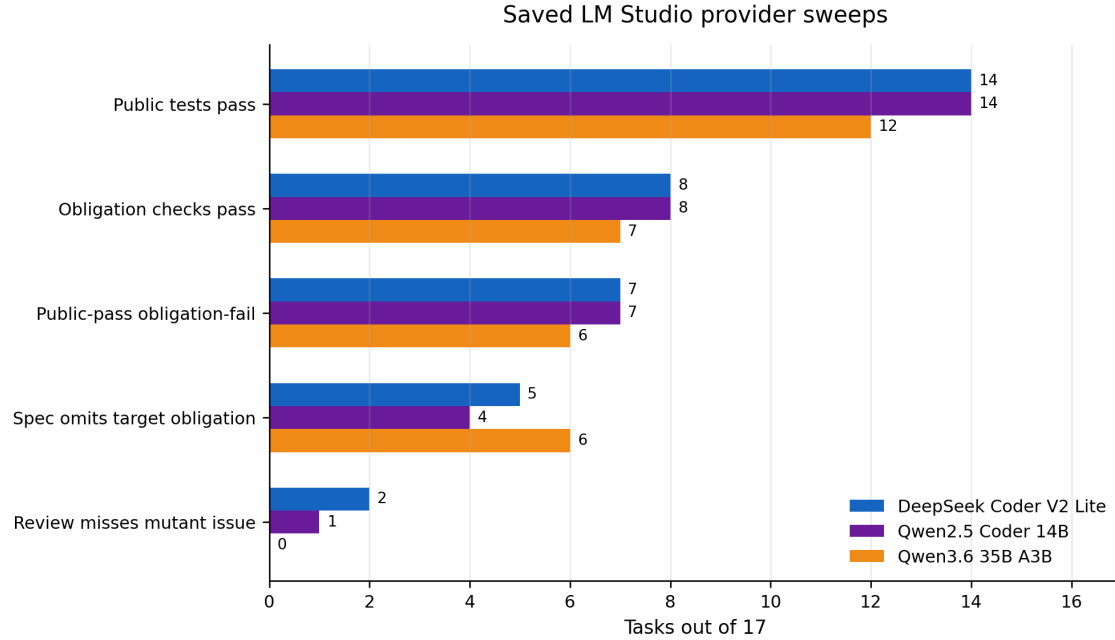


Figure 3: Saved LM Studio sweep summary. The three local models pass public tests on 12–14 of 17 tasks, but pass obligation checks on only 7–8 of 17 tasks.

8.2. Representative Failures

Privilege injection. A user deserializer defaults `is_admin` to false, but also accepts `{"is_admin": True}` from untrusted input. Public tests check defaults; the missing obligation is that privilege-bearing fields must be rejected or quarantined.

Decoded path traversal. A path joiner checks simple paths but fails to enforce containment after decoding and normalization. The counterexample escapes from `/app/data` into `/app/data_evil`, showing why string-prefix checks are insufficient.

Logout state machine. A session handler clears the visible role on logout but leaves a stale token valid. The sequence `login_admin`, `logout`, `read_secret` succeeds for the mutant, violating the temporal obligation that logout invalidates all privilege-bearing state.

8.3. Scope of Claims

The headline results use deterministic outcome checks. The saved LM Studio sweeps measure model-dependent behavior against the same fixed obligations, but they are reported as supporting evidence rather than as a general claim about all frontier models. The broader framework includes a richer preflight runner, provider-sweep infrastructure, and hooks for Hypothesis, CrossHair, and Z3. These are useful extensions, but skipped, errored, or purely abstract checks are not counted as successful implementation verification in the headline metrics.

9. Limitations, Feasibility, and Scalability of Evidence Sources

SpecGap Arena does not treat any single evidence source as decisive. Each testing class provides a different kind of assurance and fails in different ways. The value of the framework is in making these limits explicit and combining evidence against named obligations.

Table 7: Evidence sources and their limits.

Evidence	What it shows	Main limitation	Feasibility	Scalability
Public tests	Basic expected behavior on known examples.	Finite and often positive-path only; weak against missing negative obligations.	Very easy.	Cheap, but low assurance.
Obligation mutants	Reproducible false-confidence cases.	Designer-defined; not proof that every model fails this way.	Very feasible.	Scales with a curated obligation taxonomy.
Preflight templates	Turns vague prompts into explicit obligations.	Template library may miss domains or overconstrain behavior.	Feasible for security, parser, and state tasks.	Scales with modular domain tags.
Executable checks	Concrete rejection tests from obligations.	Still finite; may miss nearby variants.	Very feasible.	Easy once obligations are structured.
Contract wrappers	Makes pre/postconditions executable around code.	Runtime contracts check executions, not all paths.	Feasible for Python APIs.	Works best for narrow interfaces and pure-ish functions.
Hypothesis	Searches input families and shrinks counterexamples.	Only as strong as strategies and properties.	Feasible for Python tasks.	Good for parsers, records, and state machines.
CrossHair	Symbolic/path-based Python checking.	Python subset and contract limitations.	Medium.	Best on small pure functions.
Z3 bounded model	Exhaustive search over an abstract bounded model.	Abstraction may omit implementation behavior; bounded only.	Feasible for policy and state models.	Scales when obligations abstract cleanly.
Provider sweep	Measures model generation, spec, and review behavior.	Stochastic and provider/version dependent; saved local runs do not imply all-provider behavior.	Feasible with logging.	Scales with automation and prompt capture.
Spec diff/back-translation	Surfaces dropped or divergent obligations.	Semantic comparison is hard.	Feasible with structured schemas.	Improves with normalized obligation formats.
Boundary classification	Separates spec gaps from wrapper/runtime gaps.	Initially requires human judgment.	Feasible for case studies.	Harder, but important for real systems.

Scalable path. The scalable path is not to hand-write hidden tests indefinitely, but to grow a reusable obligation taxonomy whose entries can be emitted as tests, property checks, bounded models, and assurance cards. The framework scales horizontally by adding new task families under the same schema, and vertically by composing tasks into multi-step workflows such as parser $\rightarrow$ authorization $\rightarrow$ state transition $\rightarrow$ persistence. It is robust to LLM stochasticity because the core evidence is deterministic, while provider sweeps remain optional empirical evidence layered on fixed obligations. It is also backend-agnostic: the same obligation can be checked with executable tests, Hypothesis, CrossHair, Z3, or manual assurance-card review depending on the domain and available tooling.

10. Trusted Computing Base

The deterministic result trusts CPython execution, the benchmark harness, the human-authored obligation taxonomy, the hand-written outcome checks, and the assumption that each mutant is a plausible omission. The richer framework additionally trusts sandbox behavior, dependency availability for Hypothesis/CrossHair/Z3, provider outputs used in sweeps, and the mapping from task tags to obligation checks.

The project measures spec adequacy relative to named obligations. It does not prove full functional correctness and does not claim the obligation taxonomy is complete.

11. Conclusion

SpecGap Arena focuses on a concrete, durable subproblem: incomplete prompts can omit obligations that public tests and informal review do not expose. By using deterministic obligation mutants and a proactive obligation-preflight workflow, the project provides a reproducible way to study false confidence and a practical path for strengthening specifications before code is trusted.

References

- [1] Apart Research. The Secure Program Synthesis Hackathon, May 22–24, 2026. <https://apartresearch.com/sprints/secure-program-synthesis-hackathon-2026-05-22-to-2026-05-24>
- [2] HypothesisWorks. Hypothesis: Property-Based Testing for Python. <https://hypothesis.readthedocs.io/>
- [3] CrossHair. Analysis tool for Python. <https://crosshair.readthedocs.io/>
- [4] Leonardo de Moura and Nikolaj Bjorner. Z3: An Efficient SMT Solver. Tools and Algorithms for the Construction and Analysis of Systems, 2008.