Extracting Reliable Software from Unreliable Machines

River Keefer

LLMs are fundamentally untrustworthy.

We know it's possible to build reliable systems from unreliable components.

- Error correcting codes
- Computer networks (TCP/IP, etc)
- Consensus algorithms (Raft, Paxos, etc)
- DNA replication

Worrying about software correctness is not new.

“Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness.”

Dijkstra, “The Humble Programmer”, 1972

Some techniques I won't cover today

- Deductive verification
- Model checking
- Symbolic execution
- SMT solvers

Instead, I want to tell you about a testing methodology I think has the most hope of taming the output of these things: property-based testing.

```python
def test_can_add_users_up_to_limit():

    registry = Registry(max_users=3)

    registry.add("Alice", "alice@alice.org")
    registry.add("Bob", "bob@bob.org")
    registry.add("Charlie", "charlie@charlie.org")

    assert registry.contains("alice@alice.org")
    assert registry.contains("bob@bob.org")
    assert registry.contains("charlie@charlie.org")

    registry.add("David", "david@david.org")
    assert not registry.contains("david@david.org")
```

This test tells a story. People like thinking in stories.

But this test is falsely general.

What if we could be explicit about the range of behaviour our test is intended to cover?

```python
from hypothesis import strategies as st, given

@st.composite
def users(draw):
    name = draw(st.text())
    email = draw(st.emails())
    return (name, email)
```

```python
@given(st.data())
def test_can_add_users_up_to_limit(generator):

    # This time, we'll choose a random user cap.
    limit = generator.draw(st.integers(min_value=1, max_value=100))
    registry = Registry(max_users=limit)

    # And a random set of users.
    users = generator.draw(st.lists(users(), min_size=limit, max_size=limit))

    for (name, email) in users:
        registry.add(name, email)

    # Check that we've registered all users.
    for (_, email) in users:
        assert registry.contains(email)

    # Make sure we don't accept any further users.
    name, email = generator.draw(users())
    registry.add(name, email)
    assert not registry.contains(email)
```

PBT libraries like Hypothesis do a great job of generating adversarial data.

Some strategies include:

- Generating duplicate values
- Searching the syntax tree for literals
- Special case generators for cursed data types

Now, I want to tell you about some experiments using property-based testing as a verification layer for LLM-written code.

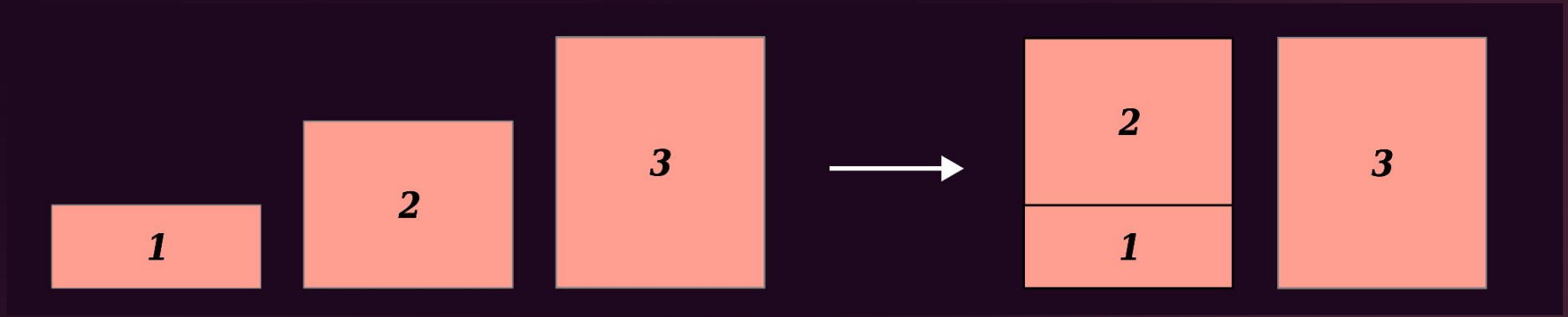
Think about the user interface of coding agents. The activity is fundamentally about writing specifications.

In fact, programming has always fundamentally been an activity of specification.

Property-based tests are a beautiful form of specification.

They don’t suffer from the imprecision of natural language, and yet they don’t require the adoption of impossibly heavyweight tools like interactive theorem provers.

First example: bin packing.



Please write a solver for the bin packing problem.
This will be used in a performance-sensitive context;
please make it run as fast as possible.

```
def pack(bin_size: int, elements: list[int]):

    """Pack ``elements`` into bins of capacity ``bin_size``.

    Uses the First Fit Decreasing heuristic, which gives a near-optimal
    bin count (within 11/9 * OPT + 1) for the offline bin packing problem.

    Returns a list of bins, where each bin is a list of the elements
    assigned to it. Every element appears in exactly one bin.

    Items larger than ``bin_size`` cannot fit in any bin and are each
    placed alone in their own (over-full) bin.
    """

    # ...
```

This gave me a fast *approximation* algorithm. That's not what I wanted.

Current LLMs are prone to filling in missing details from the natural language spec with unstated assumptions.

Property-based tests are one way of forcing yourself to make your assumptions explicit.

“Therefore, although changing to communication between machine and man conducted in the latter’s native tongue would greatly increase the machine’s burden, we have to challenge the assumption that this would simplify man’s life.”

Dijkstra, “On the Foolishness of Natural Language Programming”, 1978

```python
@settings(verbosity=Verbosity.verbose, deadline=None)
@given(st.data())
def test_agreement(generator):

    # First generate a random bin size.
    bin_size = generator.draw(st.integers(min_value=1, max_value=10))

    # Then generate a random set of items less than or equal to that size.
    elements = generator.draw(st.lists(st.integers(min_value=1, max_value=bin_size), max_size=15))

    # Ensure the fast implementation agrees with the reference implementation.
    assert pack(bin_size, elements) == reference(bin_size, elements)
```

Self-referential properties

- Function inverses (e.g. serialization and deserialization, etc)
- Idempotence (e.g. setting a value in a key/value store)
- Monotonicity (e.g. no information loss in an immutable database)

Second example: serialization

Please write a function that serializes the User class to a string, without using the json module.

```python
def test_serialization_round_trip(generator):

    # Generate some random user data.
    name = generator.draw(st.text())
    email = generator.draw(st.emails())
    user = User(name, email)

    # Serialization and deserialization should be inverses.
    assert deserialize(serialize(user)) == user
```

Third example: have the LLM write the properties

rust: remove unused actors on transaction completion #1400

Merged **alexjg** merged 2 commits into `main` from `fix-patchlog-mismatch-on-merge` last week

Conversation 0 | Commits 2 | Checks 11 | Files changed 8

alexjg commented last week — Collaborator

Problem: In issue #13900 a bug is reproduced where merging a document after making a no-op change to it panics complaining that the patch log doesn't match.

To understand this we need to know something about the `PatchLog`. The `PatchLog` carries around a bunch of operations which can be turned into patches by passing them to `Automerge::make_patches`. This avoids materializing paths and whatnot until the caller wants them. The operation IDs are lamport timestamps which refer to actor IDs via an *index* into a `Vec<ActorId>` on the document. The `PatchLog` isn't part of the document though and this means that the actor indexes in the document can get out of sync with the indexes in the `PatchLog`. To deal with this the `PatchLog` also has a `Vec<ActorId>` on it and whenever we pass the `PatchLog` to the document we add any new actors which have appeared to the `PatchLog` in `PatchLog::migrate_actors`. The actor IDs generally only ever grow, so if `PatchLog::actors` is ever larger than `OpSet::actors` we throw an error - which turns into a panic in some internal code paths.

Unfortunately there is *one* valid code path where the actor IDs array can shrink. This is when an actor is added to the document when beginning a transaction, but they are then removed because the transaction is rolled back or is a no-op. #1390 was firing because the no-op transaction added an actor to the `PatchLog` which was then not removed.

Tests like this are almost pure upside.

The worst that can happen is that we lose some time investigating a violation of a property that doesn't actually need to hold, and clarifying our spec.

An essential idea in PBT is that of *shrinking*.

```python
from hypothesis import given, strategies as st

@given(st.data())
def test_no_consecutive_three(generator):

    # Sample a completely random list of integers.
    elements = generator.draw(st.lists(st.integers()))

    # Check if the list contains three adjacent equal elements.
    for i in range(len(elements) - 2):
        assert not elements[i] == elements[i + 1] == elements[i + 2]
```

```
> pytest consecutive.py
...
    @given(st.data())
    def test_no_consecutive_three(generator):
        elements = generator.draw(st.lists(st.integers()))
        for i in range(len(elements) - 2):
>           assert not elements[i] == elements[i + 1] == elements[i + 2]
E           assert not 0 == 0
E           Falsifying example: test_no_consecutive_three(
E               generator=data(...),
E           )
E           Draw 1: [0, 0, 0]
```

Shrinking becomes even higher leverage now we can feed back shrunk test cases into the LLM loop.

Admission: these were all cases where executable specifications were relatively easy to express.

Lots of software is much messier. I think it's an open question how we grow these techniques to handle the messier cases.

Questions?

river.keefer@antithesis.com