A blog about software development.

Property-Based Testing in Rust with Arbitrary

Serhii Potapov October 21, 2022 #rust #testing #arbitrary #fuzzing

In my opinion, arbitrary is one of the most underrated Rust crates. What is so special about it? It provides the Arbitrary trait. If a type implements `Arbitrary`, then a value of that type can be obtained from `Unstructured`, which is essentially just a sequence of bytes.

This is a foundation for fuzz testing in the Rust ecosystem. Whilst fuzzing is not a common practice for every project, `Arbitrary` can be also used for property-based testing.

Property-based testing VS fuzzing

Property-based testing and fuzzing have the same idea behind them: generate random structured input and feed it to software aiming to make it break or behave incorrectly. However, fuzzing is generally a *black-box* testing method and can last for very long (e.g. months), while property-based tests rather belong to a unit test suite and may run for a fraction of second.

I find that property-based testing in some situations can be a very good replacement for classical unit tests, when we need to test symmetrical data conversion, like:

- Serialization and deserializaiton
- Converting between domain models and DTOs
- Converting between domain models and persistence layer (database records).

Introduction to the problem

Let's say we have a domain model `Vehicle` defined as the following:

```
#[derive(Debug, PartialEq, Clone)]
struct Vehicle {
    id: VehicleId,
    vehicle_type: VehicleType,
}

#[derive(Debug, PartialEq, Clone)]
struct VehicleId(i32);

#[derive(Debug, PartialEq, Clone)]
enum VehicleType {
    Car {
        fuel: Fuel,
        max_speed_kph: Option<u32>,
```

```
    },
    Bicycle,
}

#[derive(Debug, PartialEq, Clone)]
enum Fuel {
    Electricity,
    Diesel,
}
```

A vehicle can be either a car or a bicycle; a car has fuel associated with it and an optional maximum speed in kilometers per hour.

If we want to store a such model in a relational database like PostgreSQL, we'd need to convert it into a flat structure, that resembles a record in the database:

```
#[derive(Debug, PartialEq, Clone)]
struct VehicleRecord {
    id: i32,

    // Either "Car" or "Bicycle".
    kind: String,

    // Always None for Bicycle. "Electricity" or "Diesel" for Car.
    fuel: Option<String>,

    // Always None for Bicycle. Optional for Car.
    // NOTE: PostgreSQL has no unsigned integers, so we use i32 here.
    max_speed_kph: Option<i32>,
}
```

And we'll need the functions for bidirectional conversion between `Vehicle` and `VehicleRecord`.

```
fn vehicle_to_record(vehicle: Vehicle) -> VehicleRecord {
    let Vehicle { id , vehicle_type } = vehicle;
    let (kind, fuel, max_speed_kph) = match vehicle_type {
        VehicleType::Car { fuel, max_speed_kph } => {
            (
                "Car".to_owned(),
                Some(fuel_to_str(fuel).to_owned()),
                max_speed_kph.map(|speed| speed.try_into().unwrap() )
            )
        }
        Bicycle => {
            (
                "Bicycle".to_owned(),
                None,
                None,
            )
        }
    };
    let id = id.0;
    VehicleRecord { id, kind, fuel, max_speed_kph }
}

fn record_to_vehicle(record: VehicleRecord) -> Vehicle {
    let VehicleRecord { id, kind, fuel, max_speed_kph } = record;
    let id = VehicleId(id);
    let vehicle_type = match kind.as_str() {
        "Car" => {
            let fuel: String = fuel.expect("Car must have fuel");
            VehicleType::Car {
```

```
                fuel: fuel_from_str(&fuel),
                max_speed_kph: max_speed_kph.map(|speed| speed.try_into().unwrap() ),
            }
        },
        "Bicycle" => VehicleType::Bicycle,
        unknown => panic!("Unknown vehicle type: {unknown}")
    };
    Vehicle { id, vehicle_type }
}

fn fuel_from_str(s: &str) -> Fuel {
    match s {
        "Electricity" => Fuel::Electricity,
        "Diesel" => Fuel::Diesel,
        unknown_fuel => panic!("Unknown fuel: {unknown_fuel}")
    }
}

fn fuel_to_str(fuel: Fuel) -> &'static str {
    match fuel {
        Fuel::Electricity => "Electricity",
        Fuel::Diesel => "Diesel",
    }
}
```

Finally, let's implement a unit test to verify conversion from `Vehicle` to `VehicleRecord` and vice versa. If everything works correctly, we should get the same model back:

```
#[test]
fn test_vehicle_record_mapping() {
    let vehicle = Vehicle {
        id: VehicleId(123),
        vehicle_type: VehicleType::Car {
            fuel: Fuel::Electricity,
            max_speed_kph: None,
        }
    };
    let record = vehicle_to_record(vehicle.clone());
    let same_vehicle = record_to_vehicle(record);
    assert_eq!(vehicle, same_vehicle);
}
```

Even though the test is correct, it's not exhaustive and the following cases are missing:

- `fuel` is `Fuel::Diesel`
- `max_speed_kph` is present
- `vehicle_type` is `Vehicle::Bicycle`

Shall we ~~copy-paste~~ write 2-3 tests more? Or just pretend it's sufficient?

Welcome Arbitrary

Let's introduce arbitrary to the project:

```
cargo add arbitrary --features=derive
```

We also want to derive `Arbitrary` trait for our models:

```rust
use arbitrary::Arbitrary;

#[derive(Debug, PartialEq, Clone, Arbitrary)]
struct Vehicle {
    id: VehicleId,
    vehicle_type: VehicleType,
}
// And so for VehicleId, VehicleType and Fuel
```

Now we can play with it a little to get a better feeling how it works:

```rust
fn main() {
    // Some random bytes
    let bytes: [u8; 16] = [255, 40, 179, 24, 184, 113, 24, 73, 143, 51, 152, 121, 133, 1
    let mut u = arbitrary::Unstructured::new(&bytes);
    let vehicle = Vehicle::arbitrary(&mut u).unwrap();
    dbg!(&vehicle);
}
```

Output:

```
&vehicle = Vehicle {
    id: VehicleId(
        414394623,
    ),
    vehicle_type: Car {
        fuel: Electricity,
        max_speed_kph: Some(
            3870351,
        ),
    },
}
```

Is it not wonder? Out of ~~nothing~~ random bytes we have obtained a structured vehicle! If we can keep those bytes coming, we can generate an endless amount of vehicles to feed the unit test.

Rewriting the test with Arbitrary and arbtest

If we want to use `Arbitrary` for property-based testing, a tiny library like arbtest comes handy. It's a bit raw but it gets the job done. Let's rewrite our test:

```rust
#[test]
fn test_vehicle_record_mapping() {
    fn prop(u: &mut arbitrary::Unstructured<'_>) -> arbitrary::Result<()> {
        let vehicle = Vehicle::arbitrary(u)?;
        let record = vehicle_to_record(vehicle.clone());
        let same_vehicle = record_to_vehicle(record);
        assert_eq!(vehicle, same_vehicle);
        Ok(())
    }
    arbtest::builder().run(prop);
}
```

It's similar to the old test, but with a few innovations: the assertions are now within `prop` function. The `prop` function receives `Unstructured`, which is used to obtain an arbitrary vehicle. The responsibility of the builder is to generate `Unstructured` and feed it to `prop()`. In fact, `prop`

is invoked multiple times within a single test and by default, the arbtest runs it for 200ms.

OK, let's run the tests and see if it works:

```
failures:

---- test_vehicle_record_mapping stdout ----
thread 'test_vehicle_record_mapping' panicked at 'called `Result::unwrap()` on an `Err`
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


arb_test failed!
    Seed: 0x25dc50a20000003e
```

Oh no! Our bullet-proof code panics on line 49 within `vehicle_to_record()`:

```
max_speed_kph.map(|speed| speed.try_into().unwrap() )
```

`arbtest` also printed the seed `0x25dc50a20000003e`. With this seed, we can deterministically reproduce the failure and fix the test.

Let's slightly tweak our test:

```
#[test]
fn test_vehicle_record_mapping() {
    fn prop(u: &mut arbitrary::Unstructured<'_>) -> arbitrary::Result<()> {
        let vehicle = Vehicle::arbitrary(u)?;
        dbg!(&vehicle);
        let record = vehicle_to_record(vehicle.clone());
        let same_vehicle = record_to_vehicle(record);
        assert_eq!(vehicle, same_vehicle);
        Ok(())
    }
    arbtest::builder().seed(0x25dc50a20000003e).run(prop);
}
```

We added `dbg!(&vehicle);` to see which vehicle exactly causes problems and the builder is now initialized with `.seed(0x25dc50a20000003e)`, that allows us to reproduce the same failure. When we run it again, we can also see the vehicle details:

```
&vehicle = Vehicle {
    id: VehicleId(
        1455468422,
    ),
    vehicle_type: Car {
        fuel: Diesel,
        max_speed_kph: Some(
            2207965846,
        ),
    },
}
```

Having all that information, the failure becomes now very plain to explain: we use `u32` for `max_speed_kph` in the domain model, but `i32` in the DB record. And `2207965846` is above the maximum `i32` value (2^31-1).

So how do we fix it? We go to the product owner and ask a weird question.

"Hey! Do we plan to expand our system to support tracking of spaceships with speeds over 65535 km/h?"

"Oh God, no."

"Is there any possibility that we will have to support any other vehicles with speeds over 65535 km/h?" - we continue.

"Of course not! Our business is about the city's transportation network, which implies..."

"I know, I know, thank you. I just wanted to double-check."

So that means we can replace `u32` with `u16` and we're fine:

```
#[derive(Debug, PartialEq, Clone, Arbitrary)]
enum VehicleType {
    Car {
        fuel: Fuel,
        max_speed_kph: Option<u16>,
    },
    Bicycle,
}
```

We run the tests and they pass. To be extra sure we can set the time budget to 1 minute and run it locally once:

```
arbtest::builder().budget_ms(60_000).run(prop);
```

Still no failure, so we're good to go.

Conclusions

As it was shown in this article, property-based testing can be very handy for testing symmetrical data conversions. It has number of advantages over classical unit tests:

- **Much less test code needs to be written**. Especially it becomes a noticeable win, when you deal with much larger data structures.
- **The test inputs are not biased**, what is never the case when test inputs created by a human beings (aka developers).

P.S.

In real production application, we'd rather return errors, instead of just panicking in the conversions.

Someone may prefer implementing `From` traits instead of `vehicle_to_record()` and `record_to_vehicle()` functions. It's rather a question taste, but there 2 main reasons I did not go with `From`:

- I don't want `From` to panic

- I don't want the conversions to be exposed outside of the infrastructure layer.

Instead of implementing `fuel_from_str` and `fuel_to_str`, we could derive `FromStr` and `Dislpay`, using strum crate.

If you're confused about why we need both `Vehicle` and `VehicleRecord`, I'd recommend reading Domain Modeling Made Functional by Scott Wlaschin. This is out of scope for this article, but long story short: we want to keep the domain model `Vehicle` as expressive as possible, so it's easy to work with it in the domain layer. Whilst `VehicleRecord` is easy to persist in a relational database.

Links

- Discussion on Reddit
- arbitrary crate
- arbtest crate
- Fuzzing vs property testing by Ted Kaminski
- Domain Modeling Made Functional by Scott Wlaschin

Back to top
