

Formal Methods for the Working DeFi Dev

Full lecture notes: bit.ly/3RFwvBx

Rikard Hjort

Senior Verification Engineer,
Runtime Verification, Inc.



Motivation

Get the most out of your audits

How can you best prepare for an audit?

The obvious things:

- Unit tests
- Integration tests
- Documentation
- Availability

I assume these are common knowledge

But can you do more?

What's the secret sauce of a good audit?

Invariants



Find more bugs,
write better docs,
trust your code more,
help auditors.

We want you to come up with invariants of your on-chain code.

Invariants will not only help us do a better audit (because finding good invariants is what we use ourselves to find most serious issues).

They will help you find more bugs on your own.

They are good fuzz targets.

They help you view your code from a new angle.

They help you structure your code for security.

Structure

- state is just equations – how to come up with invariants describing your contract's secure state,
- proving is just ticking boxes – using simple induction to “prove” your invariants are correct – or find bugs,
- failed proofs are clues – using your invariants and proof attempts to refactor,
- invariants are test targets – how to fuzz your invariants with Foundry and,
- invariants are monitor triggers – you can use them to monitor your deployed code for inconsistencies.

If you have code you are working on right now, or have deployed, bring it up! If not, we will be working through an example, but that will be less active on your part.

It's a lot to cover. I will let this session spill out – we can always grab a table outside and keep going for another hour.



Testing vs. invariants

Testing

Testing is our first and most robust line of defense. Like a good camping knife it can get most jobs done, in the hands of a skilled technician

There is a myriad of resources to help test better, how to set up, what to write, when to write tests. And there are tools, tools, tools.

I assume you are all already somewhat well-versed in testing.



Testing is an experiment

What good is an experiment without a theory?

- Testing is great, but do you trust it to be enough?
- Fuzz-testing is all the rage. But what should you fuzz-test?

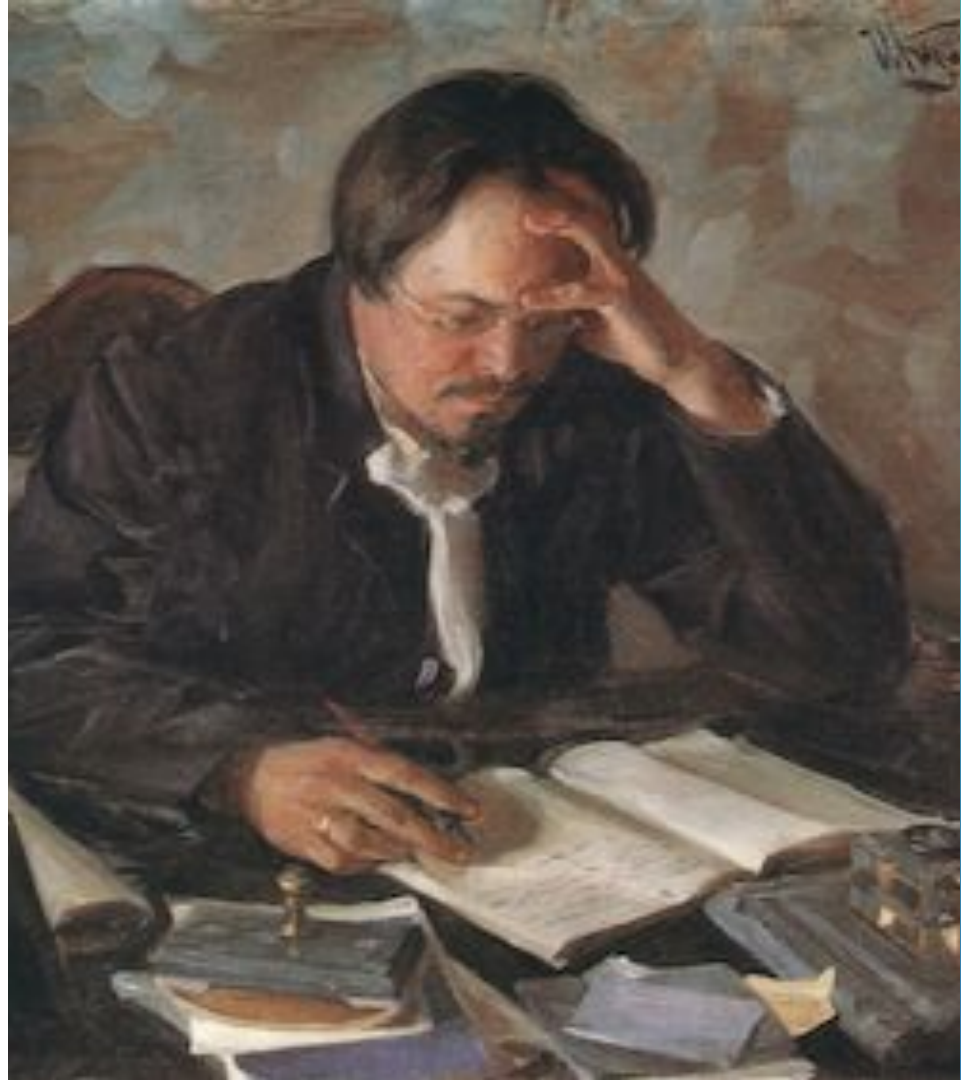
Testing is like experiments: by doing one you might learn how the system behaves in a specific instance, and with many you develop some intuition. But can we make your intuition into something rigorous, something you can communicate and rely on into the future?



Reasoning about code

- Few resources
- “You just learn from experience”
- We are always doing it:
 - Debugging
 - code review
 - Design discussions

Reasoning is like developing a theory, the thing that experiments (tests) then can help us verify.



Analogy to mathematics

In math you are taught

1. computing answers – specific results, applications of a certain rule, and how to check your answers
2. proofs – being able to prove a theorem, and being able to use it in other proofs, is the gold standard of whether you understand it.

Reasoning about code is analogous to proofs*. And just like proving (or at least hand-waving a bit to make a structured argument) there is a method to the madness, and a pinch of artistry.

*some of you may know that the correspondence between proofs and code run vary deep, but here we will take a more intuitive and less rigorous approach.

$$\begin{array}{c}
 \frac{}{\{P\}\text{SKIP}\{1 \Downarrow P\}} \text{Skip} \qquad \frac{}{\vdash_1 \{\lambda l s. P \mid (s[a/x])\}x := a\{1 \Downarrow P\}} \text{Assign} \\
 \frac{\vdash_1 \{\lambda l s. P \mid s \wedge \llbracket b \rrbracket_s\}c_1\{e \Downarrow Q\} \quad \vdash_1 \{\lambda l s. P \mid s \wedge \neg \llbracket b \rrbracket_s\}c_2\{e \Downarrow Q\}}{\vdash_1 \{P\}\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2\{e + 1 \Downarrow Q\}} \text{If} \\
 \vdash_1 \{P \mid s \wedge e'_2 s = l u\}c_1\{e_1 \Downarrow \lambda l s. Q \mid s \wedge e_2 s \leq l u\} \quad \vdash_1 \{Q\}c_2\{e_2 \Downarrow R\} \\
 \frac{\vdash_1 \{P \mid s \implies e_1 s + e'_2 s \leq e s\} \quad u \notin \text{support } P \quad u \notin \text{support } Q}{\vdash_1 \{P\}c_1; c_2\{e \Downarrow R\}} \text{Seq} \\
 \vdash_1 \{\lambda l s. I \mid s \wedge \llbracket b \rrbracket_s \wedge e' s = l u\}c\{e'' \Downarrow \lambda l s. I \mid s \wedge e s \leq l u\} \\
 \quad (\forall l s. I \mid s \wedge \llbracket b \rrbracket_s \longrightarrow e s \geq 1 + e' s + e'' s) \\
 \quad (\forall l s. I \mid s \wedge \neg \llbracket b \rrbracket_s \longrightarrow e s \geq 1) \quad u \notin \text{support } I \\
 \hline
 \vdash_1 \{I\}\text{WHILE } b \text{ DO } c\{e \Downarrow \lambda l s. I \mid s \wedge \neg \llbracket b \rrbracket_s\} \text{While} \\
 \hline
 \frac{\begin{array}{c} \vdash_1 \{P\}c\{e \Downarrow Q\} \\ \vdash_1 \{P'\}c\{e' \Downarrow Q'\} \end{array} \text{conseq} \quad \frac{\begin{array}{c} \exists k. \forall l s. P \mid s \longrightarrow e s \leq k \cdot e' s \\ \vdash_1 \{P\}c\{e \Downarrow Q\} \end{array}}{\vdash_1 \{P\}c\{e' \Downarrow Q\}} \text{const}}{\vdash_1 \{P'\}c\{e' \Downarrow Q'\}} \text{conseq}_K
 \end{array}$$



Section 3

State as Equations Invariants

What are the (incidental) rules the system must follow, if it works correctly?

invariant

- (adjective) Not varying; constant.
- (adjective) Mathematics
Unaffected by a designated operation, as a transformation of coordinates.
- (noun) An invariant quantity, function, configuration, or system.

What's the code behind the code

Your code is excruciatingly exact, and very step by step. It's imperative, not declarative.

Here's your chance to be declarative!

Example:

```
sum(balances) = totalBalance
```

A great invariant is obvious from the intended business logic, but not obvious from the code.



```
/// @dev Accounts mapped by the address that owns them.
mapping(address => Account) private _accounts;
// ...
struct Account {
    // ...
    // The share balances for each yield token.
    mapping(address => uint256) balances;
    // ...
}
// ...
/// @dev Yield token parameters mapped by token address.
mapping(address => YieldTokenParams) private _yieldTokens;
// ...
struct YieldTokenParams {
    // ...
    // The total number of shares that have been minted for this token.
    uint256 totalShares;
    // ...
}
```

Our working example

Github: [alchemix-finance/v2-foundry](https://github.com/alchemix-finance/v2-foundry) `src/AlchemistV2.sol`

Each account has a *shares* balance per yield token. We also keep track of the *totalShares* for each yield token, because we need that to calculate how much each share is worth.

So we want to make sure that these are in sync: the sum of all the balances of shares in user accounts is equal to the *totalShares*.

This invariant is crucial, because your percentage of incoming rewards are based on your percentage of the total shares.

$\forall (tokenAddress, params) \in _yield_tokens :$

$$params.totalShares = \sum_{(account) \in _accounts} account.balances[tokenAddress]$$

(This is invariant A2 in

https://github.com/runtimeverification/publications/blob/main/reports/smart-contracts/Alchemix_v2.pdf)

How to come up with good invariants?

1. Come up with many.
2. Go over your storage variables: do you have ones which have some obvious relationship?
3. **Use ghost variables:** variables that you *could* track, if you wanted, but don't. Example: `totalEthEverReceived`
4. **Deoptimize:** make a ridiculously simple version of some optimized logic. Example: distribute yields by looping over all users and updating their accounts.
5. Design spec: read your design spec again.
6. From unit tests: replace some unit tests values with symbolic values. How do the checks need to change?



Ghost variables

Feel free to introduce mathematical variables that don't correspond to any of your state variables!

This can be either impossible to compute functions, like `sum(balances)` where `balances` is a mapping, or `totalDepositedEver` which tracks all the sums that have ever been deposited into your protocol (as opposed to a `totalDeposited` state variable which may track how much is currently deposited).

You may even decide that some ghost variables deserve to become actual state variables.

Or at least, that you will emit events to be able to perform summations in your monitoring tools/indexer.

Deoptimize



```
mapping (address => uint) userStake;
uint totalStake;
mapping (address => uint) lastReward;
uint totalRewardEver;
```

// Optimized

```
function giveRewards(uint reward) {
    totalRewardEver += reward
}

function withdrawReward() {
    reward = (totalRewardEver - lastReward[msg.sender]) * userStake[msg.sender] / totalStake
    lastReward[msg.sender] = totalRewardEver
    sendTokens(msg.sender, reward)
}
```



```
mapping (address => uint) userStake;
uint totalStake;
```

// Naive

```
function giveRewards(uint reward) {
    for (uint i; i < users.length; ++i) {
        userReward[users[i]] +=
            reward * userStake[users[i]] / totalStake
    }
}

function withdrawRewards() {
    reward = userReward[msg.sender]
    userReward[msg.sender] = 0
    sendTokens(msg.sender, reward)
}
```

Another kind of invariant:

“The implementations are semantically equivalent, in terms of how many tokens get sent out”



Proving invariants:
Boring and straightforward (usually)

How to prove an invariant

Note where the (ghost) variables
are written to



Where are the variables
modified? List all functions
that modify them.

Check the “base
case”



Check that the invariant
will hold from the start:
check the constructor
and/or initializer.

Check the
“inductive case”



Prove the invariant for all the
functions/paths:

- Assume the invariant holds before the tx.
- Look at the locations that modify the variables. Perform one "symbolic update" by going through the path.
- Check that the invariant still holds

Show your work!

```
/// @dev Accounts mapped by the address that owns them.
mapping(address => Account) private _accounts;
// ...
struct Account {
    // ...
    // The share balances for each yield token.
    mapping(address => uint256) balances;
    // ...
}
// ...
/// @dev Yield token parameters mapped by token address.
mapping(address => YieldTokenParams) private _yieldTokens;
// ...
struct YieldTokenParams {
    // ...
    // The total number of shares that have been minted for this token.
    uint256 totalShares;
    // ...
}
```

Our working example

Github: [alchemix-finance/v2-foundry](https://github.com/alchemix-finance/v2-foundry)
src/AlchemistV2.sol

We look over the contract. I like to use command-line tools.

```
$ ag "balances|totalShares|assembly"
```

You need to make sure you find all relevant locations, even if they are in contracts that yours inherits, or ones that inherit from your contract.

A dependency graph tool like Surya can help.

We find:

- addYieldToken
- _issueSharesForAmount
- _burnShares

$\forall (tokenAddress, params) \in _yield_tokens :$

$params.totalShares =$


$$\sum_{(account) \in _accounts} account.balances[tokenAddress]$$

Nothing in initializer or constructor, so start at 0. The function `addYieldToken` sets `totalShares` to 0, balances untouched. We get $0 = 0$.

$\forall (tokenAddress, params) \in _yield_tokens :$

$params.totalShares =$

$$\sum_{(-, account) \in _accounts} account.balances[tokenAddress]$$



```

function _issueSharesForAmount(
    address recipient,
    address yieldToken,
    uint256 amount
) internal returns (uint256) {
    uint256 shares = convertYieldTokensToShares(yieldToken, amount);

    if (_accounts[recipient].balances[yieldToken] == 0) {
        _accounts[recipient].depositedTokens.add(yieldToken);
    }

    _accounts[recipient].balances[yieldToken] += shares; // <<<=====
    _yieldTokens[yieldToken].totalShares += shares;      // <<<=====

    return shares;
}

```

$\forall (tokenAddress, params) \in _yield_tokens :$

$params.totalShares =$

$$\sum_{(-, account) \in _accounts} account.balances[tokenAddress]$$



```
function _burnShares(address owner, address yieldToken, uint256 shares) internal {  
    Account storage account = _accounts[owner];  
  
    account.balances[yieldToken] -= shares;           // <<<=====  
    _yieldTokens[yieldToken].totalShares -= shares;  // <<<=====  
  
    if (account.balances[yieldToken] == 0) {  
        account.depositedTokens.remove(yieldToken);  
    }  
}
```

$\forall (tokenAddress, params) \in _yield_tokens :$

$params.totalShares =$

$$\sum account.balances[tokenAddress]$$

$(-, account) \in _accounts$

Done!

Find bugs, or convince everyone your invariant holds

And while this case may seem trivial, remember that

1. as things get hairier the bugs get scarier, and
2. this class of bug are definitely out there, and we do find them in frozen code.

Save somewhere useful: in
code comments, docs, design
docs ...



Failed proofs are clues,
they guide refactoring and refinement

Tools of the trade



- If the function logic **too complex**? You may need to give each basic block a name, and work it separately. Or refactor so that the same variables are modified in the same place.
- Complex arithmetic expressions, with rounding? Ignore for now, treat it as real numbers, do rounding error analysis separately.
- Working on many invariants? Annotate functions with *framing conditions*, comments which say which variables get modified in the function, both directly and indirectly.
- Did you check everything? Don't forget imports, inheritance, inline assembly...
- Code keeps changing? You may have to redo many proofs. Invariant proving is best done over frozen code.

"My head hurts and I'm not sure of my proofs!"

Look at the invariant giving you trouble. What variables does it contain?

- Pick one function of which you are not sure, look at relevant variables.
- Try to group the variable updates so that the invariant holds after each basic block. You may have to create several different paths.
- Helper functions help

Remember, *if* you show the following, you are good:

- The invariant holds at construct time/initialization
- If the invariant holds at the beginning of the ~~function~~ basic block, it holds at the end of the ~~function~~ basic block.

security > simplicity > functionality > optimization

Even if functionality and optimizations are important to you, you can start by simplifying your code, secure it by proving invariants, and gradually optimizing it while keeping your proofs up to date.



Invariants are test targets

Property based testing

You could:

- (Easy, less powerful) Create a set of tests that try a bunch of different operations, and check in the end that all your invariants hold, or,
- (Hard, more powerful) You could instrument your existing tests with some clever wrapping.



```
contract AlchemistWrapper is AlchemistV2, DSTestPlus {
    using Sets for Sets.AddressSet;
    Sets.AddressSet private users;
    // Or go hardcore, and list all functions that
    // touch *any* address.
    function transferDebtV1
        (address owner, int256 debt) public override {
        users.add(owner);
        super.transferDebtV1(owner, debt);
        checkInvariants();
    }

    // ...

    function checkInvariants() internal {
        checkA2();
        // ...
    }

    function checkA2() internal {
        for (uint i; i < _supportedYieldTokens.values.length; ++i) {
            address yieldToken = _supportedYieldTokens.values[i];
            uint sum;
            for (uint j; j < users.values.length; ++j) {
                address usr = users.values[j];
                sum += _accounts[usr].balances[yieldToken];
            }
            assertEquals(_yieldTokens[yieldToken].totalShares
                , "A2 violated");
        }
    }
}
```

Dead simple idea

Create a contract that just wraps the external functionality of the contract you are testing.

Your wrapper updates ghost variables and checks invariants upon entering or exiting the function.

Use the wrapper instead of your regular contract in testing.

$$\forall (tokenAddress, params) \in _yield_tokens : \\ params.totalShares = \\ \sum_{(-, account) \in _accounts} account.balances[tokenAddress]$$

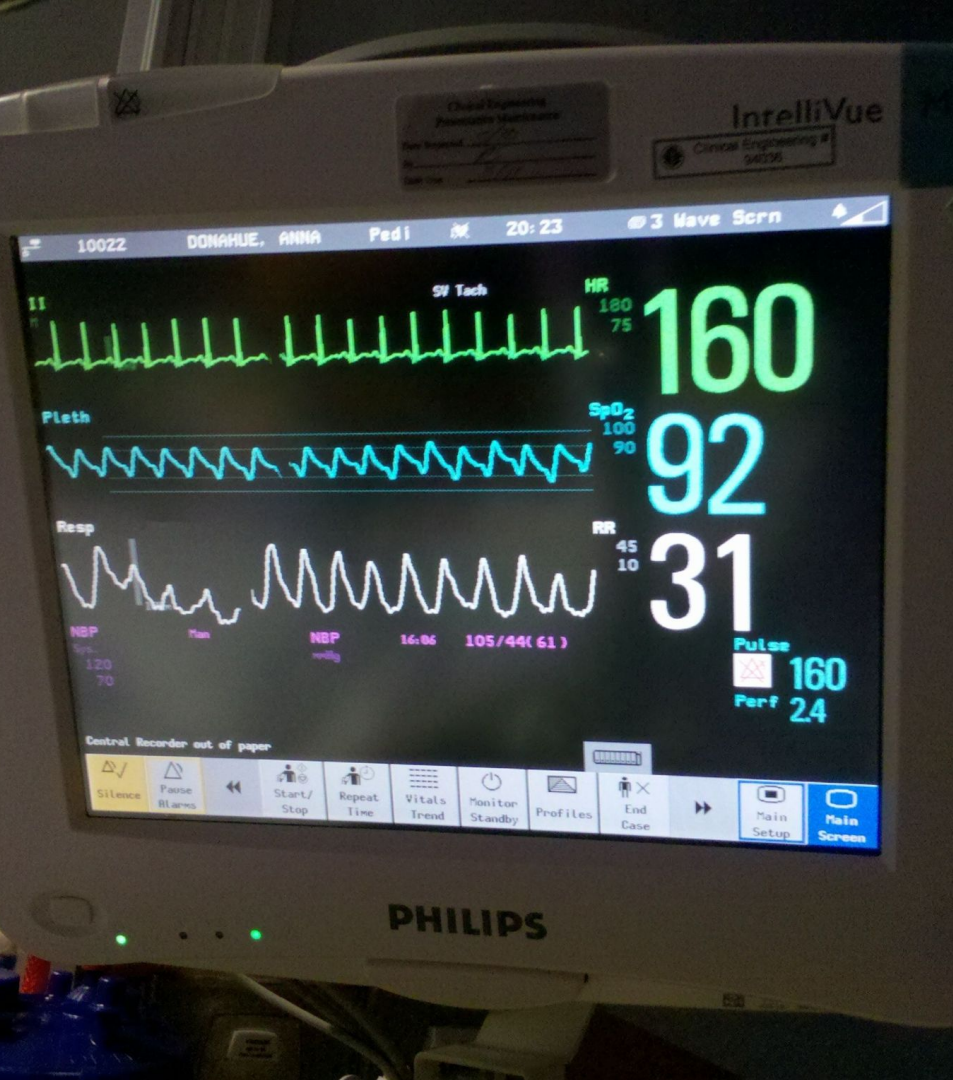


Invariants are monitor targets

Two approaches

- On-chain monitoring with requires and assert
 - Pro: can halt contract before bad things happen.
 - Con: may cause liveness bugs, requires very careful planning.
 - Con: gas cost.
- Off-chain monitoring, running your own node.
 - Pro: can be deployed at any time.
 - Pro: can't cause liveness bugs
 - Con: what do you do if you detect failure?
 - Con: may not prevent hacks that happen in a single block.

A “poor-man’s monitor” is just your Foundry fuzzing, running constantly against chain state.





What's next?

Fuzzing ♥ Symbolic Execution

Fuzzing with Foundry	Symbolic Execution with KEVM
Write test using Solidity	Reuse Foundry tests
Expressiveness limited to Solidity	Enhanced expressiveness with K-language
Extremely fast	Slow
No human intervention required	Sometimes requires human intervention
Randomized inputs	Symbolic Inputs = 100% input coverage
No false positives	No false positives
False negatives	No false negatives
Easy to use	Easy to try, hard to master

Foundry ♥ KEVM2



```
$ forge test  
[: ] Compiling...  
No files changed, compilation skipped  
  
Running 1 test for foundry-specs/TwoWayRounding.t.sol:TwoWayRounding  
[PASS] testTwoWayTrade(uint256) (runs: 256,  $\mu$ : 48595,  $\sim$ : 56824)  
Test result: ok. 1 passed; 0 failed; finished in 31.13ms
```



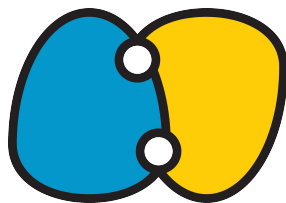
```
$ kevm foundry-kompile out  
$ kevm foundry-prove out  
Result for TWOWAYROUNDING-BIN-RUNTIME-SPEC-testTwoWayTrade:  
#Top
```



research.[runtimeverification.com](https://research.runtimeverification.com)

Past talk,
video soon

**Raoul Schaffranek: Tackling Rounding Errors
with Precision Analysis**
Oct 11th — 15:00 AM - 15:30 PM @ Flower



Questions?



<https://runtimeverification.com/>



@rikardhjort @rv_inc



<https://discord.com/invite/CurfmXNtbN>



{rikard.hjort, contact}@runtimeverification.com

Appendix: Questions from the session

- Q: What about silly edge cases, like “this [counter](#) may overflow after 2^{256} operations?”
 - A: Add the assumption that there will never be 2^{256} operations ever performed to your assumptions and move on. If you need convincing: the counter would also, incidentally, find every private key on Bitcoin, Ethereum, and all other networks. And if you processed one tx per [Planck time](#) it would take you [2 * 10²⁶](#) years to cause an overflow.
- Q: Is this anything like [Behavior-Driven Development](#), or [Given-When-Then](#) practices?
 - I have no idea! That sounds cool, tell me more about it.
- Shouldn't you do this *even if* your code changes? If you start making and proving invariants right away, won't your code be better, even if it takes a bit longer.
 - Yes, amazing idea, please do that. I just know from experience it's hard enough to get developers to start doing this at all, and it's more demoralizing if your code keeps changing under you. But if you have the discipline and good tooling go for it. Also show me your work, I would love to see what invariants you come up with.