Exposé Master Thesis
Checking Assertions for Smart Contracts

Peter Thiemann

June 14, 2021

1 Introduction

Every computation performed by a Smart Contract on the blockchain generates costs. Each unit of computation and each unit of storage used by an algorithm must be paid for. To avoid this cost, an application might perform some computation away from the blockchain (i.e., off-chain) and submit the result as a parameter to a contract on the blockchain. Typically, such a computation asserts certain properties of the submitted parameter.

However, this approach raises the issue that on the one hand the contract should take advantage of the offchain computation and assume that the submitted parameter has these properties, but on the other hand, the off-chain computation might be wrong and submit illegal parameters. So, we need a mechanism that checks the validity of the assumptions before the contract starts executing.

As an example, consider a contract that takes a prime number as a parameter.

```
contract Example {
  function (int p) public {
    // assume p is prime
    ...
  }
}
```

This assumption can be expressed with an explicit assertion in predicate logic.

$$(\forall n)(2 \leq n \leq \sqrt{p}) \Rightarrow (p \,\%\, n) \neq 0 \tag{1}$$

To test the validity of this assumption requires a loop in the contract, but the test would take $O(\sqrt{p})$ time (assuming constant time for computing the remainder) and thus produce extra cost linear in $\sqrt{p}$ accordingly. However, we could do better by recruiting the validators of the contract for a distributed effort to find a counterexample. To this end, we consider the negation of the assertion.

$$(\exists n)(2 \leq n \leq \sqrt{p}) \wedge (p \,\%\, n) = 0 \tag{2}$$

This assertion can be checked pointwise by having each validator independently choose a random n fulfilling $2 \leq n \leq \sqrt{p}$ and checking whether $(p \,\%\, n) = 0$. If the remainder is 0, the validator found a counterexample, posts its veto to the P2P net, and stops further exection. Otherwise, it accepts p knowing that other points will be checked by other validators.

In this scenario, each validator only needs to be paid to generate a random number and perform a division, which is a constant cost independent from p.

Of course, this validation is only probabilistic, so its effectiveness depends on the number of validators. One could say that the community of validators implements a Bloom filter for the set of primes: if a value p is rejected it is certainly not a prime (because there exists a counterexample); if a value p is not rejected it is prime with a probability that depends on p and the number of validators.

As another example, consider a contract that takes a sorted array of integers.

```
contract Sorted {
  function find (int[50] a, int v) public {
    // assume a is sorted
  }
}
```

The explicit assertion would be

$$(\forall k)(0 \leq k < 49) \Rightarrow a[k] \leq a[k+1] \tag{3}$$

While we can check this contract in $O(1)$ time, the constant factor is big! So we consider its negation.

$$(\exists k)(0 \leq k < 49) \wedge a[k] > a[k+1]$$

Again, we can have every validator generate a random number k. If the condition is true for such k, then the validator found a counterexample for the sortedness of the array. Otherwise, the validator relies on the other validators to check different numbers.

To obtain an estimate of the number of validators needed to find a counterexample with high probibility, let's assume the array is unsorted only at position 0, the size of the array is n, and the number of validators is m. Each validator independently has a probability of $1/n$ to detect the problem and thus probability $\frac{n-1}{n}$ not to detect the problem. Hence, if we assume that each k is chosen independenty from a uniform distribution, the probability that no validator checks at position 0 converges to 0 as the number of validators approaches infinity.

$$\lim_{m\to\infty} \frac{(n-1)^m}{n^m} = \lim_{m\to\infty} \left(\frac{n-1}{n}\right)^m = 0$$

In Dafny (citation) `https://rise4fun.com/Dafny/tutorialcontent/guide#h29` you can write

```
forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
```

to express that an array is sorted. This predicate is equivalent to the one given in (3), but it might be more challenging to test. Its negation is

$$(\exists j,k)(0 \leq j < k < |a|) \wedge a[j] > a[k] \tag{4}$$

So we'd have to generate two random numbers j and k such that the condition $0 \leq j < k < |a|$ is fulfilled.

Another condition that might be tested on an array is the heap condition

$$(\forall i)(0 \leq i < \lfloor |a|/2 \rfloor) \Rightarrow a[i] \leq a[2i+1] \wedge (2i+2 < |a| \Rightarrow a[i] \leq a[2i+2]) \tag{5}$$

2 Tasks

2.1 Offline Design

- Formally define a set of logical formulas that are amenable to the kind of checking described above. In a first approximation, the interesting assertion formulas seem to be those with prenex universal quantification.

  In a first iteration, we want to restrict ourselves to formulas of predicate logic with prenex universal quantification only.

$$\begin{aligned}
t &::= \text{primitive types like int, bool, string, } \ldots \\
\Theta &::= \Phi \mid \forall(x : t)\Theta \\
\Phi, \Psi &::= \neg\Phi \mid \Phi \wedge \Psi \mid \Phi \vee \Psi \mid M \rho N \\
\rho &::= < \mid > \mid \leq \mid \geq \mid = \mid \neq \mid \ldots \\
M, N &::= x \mid c \mid M \oplus N \mid f(\overline{M}) \\
\oplus &::= + \mid - \mid * \mid / \mid \% \mid \ldots \\
c &::= \text{constants: numbers, strings, etc} \\
f &::= \text{operations (existing in the blockchain VM)}
\end{aligned}$$

  Later, we might consider existential quantification and user-defined functions (for example, sqrt, length of a list, access list element are not predefined by the VM and have to be implemented. The list operations would be really good to have, though). See Section 3.2 for suggestions for a concrete syntax.

- Collect examples for interesting formulas.

- Formalize the cost incurred by checking a formula.

- Extend the contract language with assertions stating these logical formulas. There are several approaches to perform this extension.

  1. (Solidity) Extend the syntax of the assert statement. Requires an extension of the compiler.

2. (Tezos/Solidity) The assertion is stated in its own syntax in a file separate from the contract implementation. Requires a dedicated compiler of assertions to Michelson. The generated code is either combined with the contract implementation or stored separately on the blockchain. Several entry points may lead to difficulties.

- Implement the transformation in two steps:
  1. negation of the formula
  2. followed by the translation of an existentially quantified formula to code that picks a random value subject to predicate. For initial experiments, the target of the translation can be any programming language; later on, the target has to be a language hosted on the blockchain (e.g. Michelson or Solidity). These languages have to be extended with a random generator. There has to be a special execution mode for "challenge contracts" that search for a counterexample of an assertion.

  Negation turns prenex universal quantification into prenex existential quantification. But it would be ineffective to generate a random integer and then check that it falls in the interesting range (e.g., $2 \leq n \leq \sqrt{p}$ and $0 \leq k < n - 1$). Rather, the formula needs to be analzed and the random generator should be restricted to only generate interesting values.

  The random choice should be performed in a "clever" way that does not waste too much effort. For a wasteful example, consider (4):

$$(\exists j, k)(0 \leq j < k < |a|) \land a[j] > a[k]$$

  In the worst case, we would generate two arbitrary non-negative integers j and k and then check the predicate. However, the predicate $(0 \leq j < k < |a|)$ will fail almost always, so we do not get to test the interesting part of the predicate $a[j] > a[k]$.

  In the best case, the generated testing code would be equivalent to the following code fragment in Python

```python
def exists_counterexample (a : list) -> bool:
  if len(a) < 2: return False
  j = random.randint (0, len(a) - 2)
  k = random.randint (j+1, len(a) - 1)
  return a[j] > a[k]
```

  As another, simpler example consider (2) (checking for prime numbers)

```python
def exists_counterexample (p : int) -> bool:
  r = math.isqrt (p)
  if r < 2: return False
  n = random.randint (2, r)
  return (p % n) == 0
```

  Observe that in each case there is a test to eliminate trivial cases, which avoids illegal invocations of `randint`.

- Each test function should come with a formula that gives an estimate of its effectiveness, i.e., how many times do we have to run the test to find a counterexample with a given probability p. Ideally, the formula returns a lower bound on the number of tests for any p.
- What is the minimum number m of validators such that the probability of spotting an error is less than a given threshold q?
- Instead of using random numbers, is there a way to coordinate a systematic exploration of the iteration space? (Is this at all necessary?)
- What are the connections to random testing?
- I think this scheme implements a Bloom filter, globally? We are testing whether a value is a member of an implicitly given set. We may accept a value even if it does not belong to the set, but if we reject a value, it definitely does not belong to the set.

2.2 Blockchain-Specific Tasks

Ethereum-specific remarks: Implementing this scheme requires adding instructions to the virtual machine. One instruction would generate a random number (EVM uses 512 bit unsigned integer IIRC), the other would raise an overriding failure. This failure will have to indicate that validation of the contract fails.

- The VM interpreter needs to be extended with these instructions.
- The translation (of formulas) needs to retargetted to VM code. This may have to be hacked into the compiler.
- The translation has to be amended so that counterexamples can also be checked.
- (extension) Every validator could perform a predetermined finite number of checks for each assertion. This would not be significantly more expensive. Moreover, the number of checks could be set individually depending on the contract and on the size of the input.
- (extension) One could also check properties of the values stored with the contract.

2.3 Protocol Design

Tezos specific remarks: here the number of validators is really small compared to Bitcoin or Ethereum

- Maybe we need to propose and validate the same block multiple times before it gets accepted. E.g, a transaction containing this feature is only accepted if it is revalidated n times.
- Extra validators only for this kind of transaction.
- Maybe each validator needs to conduct multiple test runs.
- Maybe test runs need to be coordinated

- (Tezos) Proposal for an architecture. The assertion for contract C gets compiled to a separate contract A(C). A(C) takes C's argument x and attempts to challenge the assertion by finding a counterexample. Given x, we determine a number n of "probation cycles" during which the net attempts to find a counterexample. The injecting node broadcasts A(C)(x) for n cycles. If A(C)(x) finishes successfully, this execution vanishes (as no counterexample has been found). If A(C)(x) finds a counterexample, it records the counterexample along with the data to check it on the blockchain as in A(C,x,y) where it can be validated by the net as usual. After n cycles the injecting node broadcasts C(x), which is processed as usual. However, a valid blockchain either contains C(x) or A(C,x,y), but not both. It is an error to attempt to enter C(x) after A(C,x,y) is on the chain, and vice versa.

  It needs to be investigated which aspects of the protocol need to be adapted. Moreover, A(C,x) must execute in a different mode that enables the random instruction. This instruction is not available in the default execution mode.

- Once a counterexample has been found, this counterexample is broadcast to the network where it can be verified by further nodes. I think this broadcast and the reaction to this broadcast requires an update of the protocol: all receivers of the broadcast will check the counterexample and revert the transaction.

- (see above for a revised approach) A transaction using counterexamples has to wait for a couple of cycles in the transaction pool until the node feels safe to include the transaction in a block. The number of cycles to wait depends on the contract and on the size of the input. So that number should be variable.

- (low priority) This approach poses the danger of a DOS attack. Someone could post a false counterexample, which does not really lead to a failure.

If everyone switches to check this proposed counterexample instead of generating a random number, then the real problem may remain hidden. So, there has to be a component in the protocol that penalizes someone who broadcasts a false counterexample. Moreover, not all validators should switch to recheck a proposed counterexample; or they could check it in addition to their own random check.

More precisely:

- Every validator must run at least one (round of) random tests.
- If it receives a proposed counterexample, it must check the counterexample, too.
- It remains to investigate a countermeasure for nodes generating a flood of false counterexamples, which would keep every validator busy. Perhaps checking of the counterexample should also be done probabilistically.

3 Suggestions

3.1 General

In the example, we checked if all elements satisfy a certain condition:

```
forall (n : Int)
   if ( 2 <= n && n <= sqrt(p))
       assert(p%n != 0)
```

However, if we consider parameter p to be a sorted list and we want to assert that it is indeed sorted, we would need to check two values:

```
forall (n : Int, m: Int)
   if ( n > m)
       assert(p.(n) > p.(m))
```

You can have foralls with more than one variable, but you could also nest the foralls as in

```
forall( n : int)
 forall(m : int) ...
```

Externally, I think your proposed syntax is nicer, but internally it will be simpler to just have `Forall( ident, type, expr)` in the type for the AST.

Somehow the smart random generator will have to know in which range to generate the values. Considering the second example, the generator should generate

```
m = random(0, length(list))
n = random(m, length(list))
```

For this transformation, the assertion should be completely explicit. That is, in the example, there should be an additional condition

```
... if (n < length(p) && m < length(p)) ...
```

which means we should have the length of a list available as a primitive.

Can we realize this transformation from the above syntax, or do we need additional syntax that expresses this transformation separately? Do I even have to consider these things when creating the grammar?

The process could be something like this:

1. calculate the negation of the formula. This amounts to swapping forall and exists, skipping the conditions, and negating the body of the assertion. This last step should also be done by applying de Morgan exhaustively down to the literals. In your example you'd end up with

```
exists(n : int)
 exists(m : int)
    if (n< length(p) && m < length(p))
      if (n>m)
         check( p.(n) <= p.(m) )
```

2. look at the atomic constraints that bound the exists and move them upfront. Example:

   - the first constraint is `n<length(p)` → you need to break up the conjunction
   - independent of `exists(m)`, hence move backward

```
exists(n : int)
 if (n < length(p))
   exists(m : int)
     if (m < length(p))
       ...
```

3. if the constraint meets its generator (exists), then merge it:

```
exists(n : int, n < length(p))
```

4. same for m

```
 exists(m : int, m < length(p))
```

   (at this point, each generator / exists could come with a number of upper and lower bounds)

5. now we need to consider this

```
exists(n : int, n < length(p))
 exists(m : int, m < length(p))
   if (n>m)
     ...
```

   this condition gives us an upper bound on m (this requires some manipulation of the formula, in general) so we just add it as another bound to m:

```
exists(n : int, n < length(p))
 exists(m : int, m < length(p) && m < n)
    ...
```

6. you can find out (but this is more involved and thus optional) that `0 < n` is required for the `exists(m ...)` to be nonempty and that `m < n` implies `m < length(p)`:

```
exists(n : int, 0 < n,  n < length(p))
 exists(m : int, 0 <=m, m < n)
   check( p.(n) <= p.(m) )
```

   At this point, you can read off the invocation for the random generator. If any constraints remain, they'd have to be checked at run time. In the best case, things work out like in the example. Otherwise, each if-constaint that turns out to be false results in a useless test run.

For background, there is a nice paper about the underlying testing methodology: How to specify it! by John Hughes: `https://www.tfp2019.org/resources/tfp2019-how-to-specify-it.pdf`

3.2 Tezos

Michelson and the Tezos infrastructure directly support entry points with a specific notation. Hence, the syntax should reflect and reuse that notation.

The syntax for assertions should be chosen to be similar to OCaml. A proposal for the assertion in (1) could be

```
entrypoint %default
  parameter (_, (_, p)) =
    forall (n : int)
      if (2 <= n && n <= p / 2)
        p % n <> 0
```

A parameter phrase may occur multiple times. Its single argument is a pattern that matches the contract's entrypoint's argument. The type of the pattern variables (here just `p`) is determined from the code of the contract.

The entrypoint phrase may also occur multiple times. However, it is illegal to have patterns that overlap, i.e., which match the same contract invocation. Overlapping may occur due to a repeated pattern in clauses for the same entrypoint, but also due to a pattern for the `%default` overlapping with a pattern for another entrypoint.

In Tezos/Michelson, entry points are modeled by having a sum type as parameter type. For example, a contract with two entry points can be equipped with assertions as follows.

```
parameter (Left x) =
  0 < x && x < 100

parameter (Right (a, b)) =
  exists (c : int)
    if (0 < c && c <= a + b)
      a*a + b*b = c*c
```

BTW, this is an example for a useful contract with an existential!

These phrases would refer to the `%default` entrypoint, as none is explicitly specified. Suppose the parameter type for the contract in question was specified in Michelson as follows.

```
parameter (or (int %set_scale) (pair int int %set_triangle))
```

An equivalent way to stating the previous assertions would be to use entrypoints.

```
entrypoint %set_scale
  parameter (x) =
    0 < x && x < 100

entrypoint %set_triangle
  parameter (a, b) =
    exists (c : int)
      if (0 < c && c <= a + b)
        a*a + b*b = c*c
```

The idea would be to have two files with the same basename, but different extensions. One file contains the Michelson code, the other the assertion.

4 Rejected Ideas

This section contains some ideas that we discussed but rejected. Feel free to resurrect any of this if you find new arguments.

- Q: How about instead of generating and checking random number, it could store a Bloom filter for already checked number, so a validator can generate a random number and check it if it is rejected by Bloom filter, but this also happen that there will be some number never get checked because it is rejected by Bloom filter, but actually never been checked.

  A: This is not a good idea for several reasons.

  1. It's not clear where we should store the Bloom filter. It has to be available everywhere, so it has to be stored on the blockchain.
  2. If it's on the blockchain, then it serializes the tests for counterexamples: the point of the scheme is to test many random values concurrently without interference.
  3. The Bloom filter errs in the wrong direction. It may accept elements that have not been added before, which means that elements that should be tested are not. We would need a co-Bloom filter that errs in the other direction: it should reject all elements that have not been added, and it may also reject elements that have been added.