Examples for Interesting Formulas

Tamara Bernhardt, Julian Veigel

December 2020

1 Collection

The following list includes our whole collection of interesting formulas for distributed assertion checking. We went into more detail for some examples (marked in color) in the following chapter. The discussion, whether or not it's possible to implement assertion contracts for these examples for the Tezos and Ethereum blockchains is left out for the moment.

- **Logic**
  - CNF SAT Problem
  - DNF SAT Problem
- **Numerical**
  - Number is a prime number
  - One number is the greatest common divisor (gcd) of two other numbers
    - Special case: Two numbers are coprime, i.e., $gcd(a, b) = 1$
  - One number is the least common multiple of two other numbers
- **Arrays**
  - Array is sorted
  - Fulfills (balanced) binary search tree property, i.e., the left elements of subtree are smaller, right elements of subtree are greater than parent
    - Problem: handling trees which are not full, e.g. by defining empty elements?!
  - Fulfills heap property, i.e.
    - Max-Heap: any child is smaller or equal than its parent
    - Min-Heap: any child is greater or equal than its parent
    - Problem: handling trees which are not full, e.g. by defining empty elements?!

  - All Elements in one are bigger / smaller than in another
  - One array is the reverse of another (similar to sorted array)
- **Strings**
  - All characters are upper case, lower case, alphanumerical, ...
  - One string is (not) a substring of another string (using slices)
- **Sets**
  - Set property is fulfilled: no element occurs twice (if set is represented as an array)
  - Two sets are disjoint
- **Matrices**
  - Identity matrix $\mathbb{I}$: square matrix with only diagonal ones, else zeros
  - Logical matrix (only one or zero entries), matrix of ones (only one entries), zero matrix, ...
  - Symmetric matrix: $\mathbf{A} = \mathbf{A}^T$
  - Orthogonal matrix: $\mathbf{A}\mathbf{A}^T = \mathbb{I}$
  - Correct calculation result, e.g. matrix multiplication, multiplication with a constant, addition, ... (which might be more involved and mostly need all terms for verification, which is memory consuming)

2 Examples

2.1 CNF SAT Problem

Instances of the SAT-problem consist of a propositional logic formula F, for which it needs to be determined if there exists a variable assignment $\mathcal{A}$ to the values $True$ or $False$, s.t. the formula is satisfied ($\mathcal{A}(F) = 1$). Most SAT-solvers expect F to be in conjunctive normal form (CNF), i.e., a conjunction of a set of clauses, where clauses are disjunctions of variables or their negations (called literals).

Verifying whether a given assignment satisfies the formula takes $\mathcal{O}(n * m)$, where n is the number of clauses and m the number of variables (in the worst case, every variable is present as a literal in each clause).

2.1.1 Encoding of variables, literals and clauses

Consider a contract that takes a SAT problem formulated in CNF as a list of clauses c and a variable assignment a, s.t. the formula is satisfied. To avoid having to handle the variables and their literals as strings and thus optimize runtime and memory consumption, we use an encoding scheme taken from this blogpost [2]:

Variables are encoded by a unique number starting from 0 to $n-1$, where n is the number of variables. The positive and negative literals of a variable encoded by x is encoded by the function

$$encode(x) = \begin{cases} 2x & \text{if x is a positive literal} \\ 2x+1 & \text{if x is a negated literal} \end{cases}$$

Let's consider the following example:

$$F = (A \vee B) \wedge (B \vee \neg C),\ \mathcal{A} = \{A \to False, B \to True, C \to False\}$$

The list of variables $v = [A, B, C]$ occurring in F is encoded as $v_{enc} = [0, 1, 2]$. We've already considered F to be a list of clauses, but we can also look at clauses as a list of literals. After encoding the literals, F is represented as $F_{enc} = [[0, 2], [2, 5]]$. Since the encoding of variables corresponds to their index in the list of variables, we can simply pass $\mathcal{A}$ as a list of booleans: $\mathcal{A}_{enc} = [False, True, False]$. Alternatively, $\mathcal{A}$ can be represented as a bitarray to reduce the memory consumption further.
Checking whether a literal is positive or negative is a simple bit-wise AND operation:

```
isPositive(x) = x & 1 == 0
```

Looking up the variable of a literal is done by dividing the encoded literal by two, which corresponds to a bit-wise shift to the right:

```
decode(x) = x >> 1
```

2.1.2 Formulation in predicate logic

Given that F and $\mathcal{A}$ are encoded like this, the assumption that $\mathcal{A}$ satisfies F can be expressed in predicate logic as follows:

$$(\forall i, \exists j)(0 \leq i < |F|) \wedge (0 \leq j < |F[i]|) \Rightarrow ((isPos(F[i][j]) \wedge \mathcal{A}[decode(F[i][j])]) \vee (\neg isPos(F[i][j]) \wedge \neg\mathcal{A}[decode(F[i][j])]))$$

The transformed formula to check for any counterexamples is then:

$$(\exists i, \forall j)(0 \leq i < |F|) \wedge (0 \leq j < |F[i]|) \wedge ((isPos(F[i][j]) \wedge \neg\mathcal{A}[decode(F[i][j])]) \vee (\neg isPos(F[i][j]) \wedge \mathcal{A}[decode(F[i][j])]))$$

2.1.3 Assertion contract

The corresponding assertion contract for this example is shown in listings 1 and 2:

Listing 1: Michelson contract signature

```
parameter (pair (list int %f)
                (list bool %a))
```

Listing 2: Assertion contract

```
(entrypoint (pair (f: (list int))
                  (a: (list bool)))

  (forall(i: int)
    (if (and (ge i 0) (lt i (size f)))
      (exists(j: int)
        (if (and (ge i 0) (lt i (size (nth f i)))))))
          (assert
            (if (eq (and j 1) 1)
              (nth a (lsr j 1))
              (not ((nth a (lsr j 1))))))))
```

2.2 DNF SAT Problem and Proofs of Correctness

SAT problems can also be formulated using the disjunctive normal form (DNF), a disjunction of clauses, which consist of conjunctions of literals. At first glance, one might think handling this case is just a matter of swapping the two quantifiers, i.e.

$$(\exists i, \forall j)(0 \leq i < |F|) \wedge (0 \leq j < |F[i]|) \Rightarrow ((isPos(F[i][j]) \wedge \mathcal{A}[decode(F[i][j])]) \\ \vee (\neg isPos(F[i][j]) \wedge \neg \mathcal{A}[decode(F[i][j])]))$$

However, this formula raises an interesting problem for distributed assertion checking. Let's look at the transformation of the formula:

$$(\forall i, \exists j)(0 \leq i < |F|) \wedge (0 \leq j < |F[i]|) \wedge ((isPos(F[i][j]) \wedge \neg \mathcal{A}[decode(F[i][j])]) \\ \vee (\neg isPos(F[i][j]) \wedge \mathcal{A}[decode(F[i][j])]))$$

Due to the disjunction, finding two random numbers i, j for which the transformed formula evaluates to false will produce a false counterexample - more so, it will actually produce a proof of correctness! In that case, the caller of the contract would need to wait until it receives a proof from one of the validators. This raises an important question: how can the caller differentiate between the two cases, i.e., waiting for a counterexample vs. waiting for a proof? In the cases of single quantifiers, the answer is trivial. Existential quantifiers will require a proof, whereas universal quantifiers require a counterexample. However, it's less obvious for nested quantifiers, especially since nothing is stopping the programmer of an assertion contract from swapping the order of quantifiers. Currently, the following two cases are not handled differently in our transformation:

```
(forall(i: int)
  (exists(j: int)
    ... ))

(exists(j: int)
```

```
(forall(i: int)
  ... ))
```

Things we need to clarify/consider:

- Is there a way to differentiate between requiring a proof/counterexample in case of nested quantifiers?
- If not:
  - Does a defined order of quantifiers fix this?
    - Responsibility of programmer?
    - Can we even enforce a "correct" order of quantifiers?
  - Add explicit directives 'proof' and 'counterexample'?
    - Could this replace "forall" and "exists" (see listing 3)? Does it make writing the assertion less intuitive?

Listing 3: Alternative syntax using proof or counterexample expressions

```
proof(n: int, m: int) ...

(proof
  (generate (n: int)
    (generate (m: int)
      ... )))
```

2.3 Two numbers are coprime

The greatest common divisor (*gcd*) of two non zero integers is the largest integer that divides both evenly [3]. A more specific *gcd*-problem is to find two numbers which are coprime, also called relative prime. Coprime expresses the relation between two integers a and b, where the greatest common divisor between those two is one [4], i.e., $gcd(a, b) = 1$. To check if two numbers are coprime it has to be ensured that there is no greater *gcd* than one, which can be stated in a propositional formula.

$$(\forall n)(2 \leq n \leq \min(a, b)) \Rightarrow \neg((a \,\%\, n) = 0 \wedge (b \,\%\, n) = 0) \tag{1}$$

To check this it would take $\mathcal{O}(a)$, means linear time complexity. To find a contradiction to this condition we can formulate:

$$(\exists n)(2 \leq n \leq \min(a, b)) \wedge (a \,\%\, n) = 0 \wedge (b \,\%\, n) = 0 \tag{2}$$

When it is ensured that the greater number is not evenly divided by the smaller one ($\max(a, b) \,\%\, \min(a, b) = 0$), which automatically would then be a counterexample, the search space can reduced by half to ($2 \leq n \leq \lfloor \frac{a}{2} \rfloor$), because in the upper half of the lower number no integer can evenly divide this number.

In order to take advantage of this, the assertion syntax would need to be able to create different generators for different conditions. Currently, we only allow `if-else` instructions inside an `assertion` expression; this use-case could be an argument to lift that restriction and allow contracts like this:

Listing 4: Assertion contract with conditional generators

```
(entrypoint (pair (a: int) (b: int))
  (if (lt a b)
    (forall (n: int)
      (if (eq 0 (mod b a))
        (* then *)
        (if (and (le 2 n)
                 (le n (div a 2)))
          (assert
            (not (and (eq (mod a n) 0)
                      (eq (mod b n) 0)))))
        (* else *)
        (if (and (le 2 n)
                 (le n a))
          (assert
            (not (and (eq (mod a n) 0)
                      (eq (mod b n) 0)))))))))
```

However, this will make the transformation more involved, as the bounds cannot simply be merged with the generator anymore.

2.3.1 Least common multiple

A similar problem would be to check the least common multiple (lcm) of two numbers. There the space between the greater number and the stated lcm has to be checked for a smaller one. This search space can be significantly reduced by only selecting numbers, which are multiples of the greater number and check if the smaller number can divide this number without remainder.

$$(\exists n)(2 \leq n < \frac{lcm}{\max(a,b)}) \wedge ((\max(a,b) * n) \,\%\, \min(a,b) = 0) \tag{3}$$

2.4 Array fulfills heap property

There are two categories of heaps: min and max heaps. Both can be represented as binary trees. In a max heap, any given node has a lower or equal value than the value of its parent. In a min heap, the value is greater or equal [5]. Binary heaps are often implemented as an array. Given an array representation of the tree according to Eytzinger's layout [1], the parents and children of a node can be accessed as follows: [6]

```
heap_array[(k-1)/2] // Get parent of actual node k
heap_array[(k*2)+1] // Get left child of actual node k
```

```
heap_array[(k*2)+2] // Get right child of actual node k
```

Listing 5: Access a heap in array representation

So to check if an array a fulfills the min heap property, we can formulate it in the following predicate logic:

$$(\forall k)(1 \leq k < |a|) \Rightarrow a[\lfloor (k-1)/2) \rfloor] \leq a[k] \tag{4}$$

This equation checks the direction from the child to the parent node. (Following the Exposé) The direction could also be turned around and every parent node could be checked against its child nodes, which does not change the number of comparisons.

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

This test would need $\mathcal{O}(|a|)$, i.e., linear time complexity. To find a contradiction this can be formulated to

$$(\exists k)(1 \leq k \leq |a|) \wedge a[\lfloor (k-1)/2) \rfloor] > a[k] \tag{6}$$

This example is similar to the example for checking if an array is sorted. Further such examples could be:

- One array is the reverse of another
- Array fulfills binary search tree property (If not balanced, empty fields must be identified correctly)

2.5 String is alphanumeric

The characters used in an input string s might need to fulfill different properties. For instance, all characters of a string must be alphanumeric, or expressed in regex each character must match `[a-zA-Z0-9]` $\equiv$ `[[:alnum:]]`. Similarly to checking every element of an array, this can be checked with a linear $\mathcal{O}(|s|)$ time complexity. To formulate this property we can state the logic

$$(\forall n)(0 \leq n < |s|) \Rightarrow alnum(s[n]) \tag{7}$$

The negation of this formula for finding a counterexample can also easily be stated as

$$(\exists n)(0 \leq n < |s|) \wedge \neg alnum(s[n]) \tag{8}$$

Where

$$alnum(x) = \begin{cases} \bot & \text{if } x \text{ is not alphanumeric} \\ \top & \text{if } x \text{ is alphanumeric} \end{cases}$$

Note that more complicated regular expressions involving operators like repetition or option are not feasible, since they require state, which is not compatible with our random testing approach.

2.6 No intersection between two sets

To check if two sets u and v have no common subset, every element of one set must be compared to all other elements of the other set. This comparison leads to a complexity of $\mathcal{O}(|u| * |v|)$. We furthermore can define the size of both sets together as n, i.e., $n = |u| + |v|$. The worst case for the run time would be that both sets have the same size, i.e., $|u| = |v| = n/2$. So we can see that this leads to a quadratic complexity $\mathcal{O}(n/2 * n/2) = \mathcal{O}(n^2/4) = \mathcal{O}(n^2)$. The formula to state this property is

$$(\forall i, \forall j)(0 \leq i < |u|) \wedge (0 \leq j < |v|) \Rightarrow u[i] \neq v[j] \tag{9}$$

To check this property the formula can be translated to existential quantification:

$$(\exists i, \exists j)(0 \leq i < |u|) \wedge (0 \leq j < |v|) \wedge u[i] = v[j] \tag{10}$$

Further such examples could be:

- All elements in one array are bigger than in another

3 Estimate of effectiveness

What remains to be analyzed is how many test runs are needed for a formula in order to reach a specified certainty threshold. This is especially important for Tezos, where the number of active validators per cycle is rather small. To compensate for this, each validator might need to execute several test runs.

3.1 Approach 1: Coupon Collector Problem

A simple, yet naive first approach on finding a formula that gives the number of test runs, is to set it to the size of the search space $|\mathcal{S}|$. However, the probability of finding a counterexample is low, since the generated random numbers will most likely contain duplicates and thus not cover the entire search space. Given $|\mathcal{S}| = n$, the probability that no duplicates are generated is given by

$$p = \prod_{i=1}^{n} \frac{i}{n}$$

For $n = 10$ the probability already drops to 0.036%.

This is called the "Coupon collector's problem"[7] in probability theory. Let T be the number of test runs that are executed until every element in the search space has been generated. The goal is to identify its expectation $E(T)$. For this, we apply the geometric probability distribution:

Since our random generator generates elements with an equal distribution, each number is generated with a probability of $1/n$. The probability to generate the ith new number is given by

$$p_i = \frac{n - i + 1}{n} \tag{11}$$

The expected value of a random variable X is given by $E(X) = \frac{1}{p}$[8], thus the expected number of test runs for n is

$$E(T) = n \sum_{i=1}^{n} \frac{1}{i} \tag{12}$$

The following table 3.1 shows the expected number of test runs $E(T)$ for different sizes of $\mathcal{S}$.
The variance of this value is given by

$$\sigma^2 = \sum_{i=1}^{n-1} \frac{i}{n-1} \tag{13}$$

from which the standard derivation σ can easily be obtained by taking the square root. Some values for σ are shown in table 3.1. Having determined the expected value and the standard derivation, the *central limit theorem* can be used to calculate a lower and an upper bound for the number of needed test runs. A 95% confidence interval is found by adding $1.96 * \sigma$ on both sides of $E(T)$[7], as shown in table 3.1.

n	$E(T)$	σ	lower bound	upper bound
2	3.0	1.0	1.04	4.96
5	11.4	2.53	6	16
10	29.3	19.29	21	38
20	71.95	7.21	58	86
50	224.96	13.23	199	251

Table 1: $E(T)$ for search spaces of different sizes n

This method targets a 100% coverage of the search space, e.g. it obtains the number of test runs to find a counterexample with probability $p \approx 1$. Section 3.2 describes an approach to calculate the number of test runs necessary to reach a specified probability threshold.

3.2 Approach 2: Achieving probability thresholds

As in the previous chapter defining the search space $\mathcal{S}$ as the set of all test runs, therefore any potentially existing counterexample is in this set. The size of this set is defined as $n = |\mathcal{S}|$, which is the number of test runs to test all possibilities. For this calculation it is assumed that there exist exactly one counterexample in the complete search space. Goal is to find a number t of test runs (which should ideally be smaller than the size of the search space n), s.t. the probability $P_{c,t}$ of not finding the counterexample drops below a certain threshold c.

The probability of finding the counterexample with one test run is n^{-1}, and of not finding the counterexample $(1 - \frac{1}{n})$. The probability of t test runs not finding the counterexample is $(1 - \frac{1}{n})^t$.

Making small changes to the Bernoulli's inequality approximation can be used for further calculation [11]:

$$(1-\frac{1}{n})^t = ((1-\frac{1}{n})^n)^{\frac{t}{n}} \leq e^{-\frac{t}{n}} \tag{14}$$

because

$$m > 1 : (1-\frac{1}{m})^m \leq e^{-1}$$

So we can define the probability $P_{c,t}$ of not finding the counterexample with t test runs [12]:

$$P_{c,t} \leq e^{-t/n} \tag{15}$$

Now we can check how many test runs are necessary that this probability drops below a certain threshold c:

$$\begin{aligned} e^{-t/n} &\leq c \qquad\qquad \text{with } c \leq 1 \\ t &\geq -n \ln(c) \end{aligned} \tag{16}$$

For $c = e^{-1}$ ($\approx 0.36788 = 36.788\,\%$) t is exactly n. So for every threshold below this value there are more test runs needed than there are possibilities in the search space itself.

3.3 Tezos Bakers

In the current cycle (311) there are 418 registered bakers [9]. However, not every baker that has been granted rights for baking are also actively making use of that right. According to [10], only 398 of the 418 bakers have participated in the current cycle.
Theoretically, every assertion formula with $E(T), t \leq$ #active bakers can be verified with a single test run per validator (baker), assuming that **all** validators execute the test run after receiving a request. If this assumption cannot be made, it might be necessary to request more test runs per validator, if $E(T)/t$ is too close to the number of active bakers. However, this conversely means accepting a potentially high overshoot of test runs (this is also the case for values of $E(T)/t$ slightly above the number of bakers).

3.4 Results for the given examples

For the first example, we provide full calculations with a given input size using both approaches for comparison. Since the calculations for the other examples will not differ, we only state the sizes of their search spaces.

3.4.1 CNF SAT Problem

In the case of the CNF SAT problem, at least one variable in every clause needs to be checked to verify the correctness of the assignment. Thus, the size of the search space is the number of clauses, i.e., $n = \#clauses$.

Let's assume a CNF with 20 clauses, i.e., $n = 20$. Using approach 1, we calculate the expected number of test runs:

$$E(T) = 20 \sum_{i=1}^{20} \frac{1}{i} = 71.95$$

As explained above, this will give us $p \approx 1$ that a counterexample is found if one exists. Using equation 16 to calculate the same, we set $c = 0.01$ (as $c = 0$ will result in infinity) to achieve $p = 0.99$:

$$t \geq -20 \ln(0.01) = 92.1$$

A certainty of e.g. 90% of finding a counterexample if one exists is achieved by

$$t \geq -20 \ln(0.1) = 46.1$$

3.4.2 DNF SAT Problem

Similarly to CNF, the search space is also given by the number of clauses, i.e., $n = \#clauses$.

3.4.3 Coprime numbers

The size of the search space is defined by the smaller of the two numbers (named a in the formula given above), namely

$$n = \begin{cases} a - 1 & \text{if } b \,\%\, a = 0 \\ \lfloor \frac{a}{2} \rfloor - 1 & \text{if } b \,\%\, a \neq 0 \end{cases}$$

3.4.4 Array heap property

The size of the search space equals the size of the array without the root node, i.e., $n = |a| - 1$.

3.4.5 String is alphanumeric

The size of the search space equals the size of the string, i.e., $n = |s|$.

3.4.6 Disjoint sets

In this case, the number of test runs can't simply be derived from a single search space, but both set sizes must be considered. In practice, the sets most likely won't be of the same size; therefore we must chose the size of the bigger set to make sure no element of the smaller set is included:

$$n = \begin{cases} |u| & \text{if } |u| \geq |v| \\ |v| & \text{if } |u| < |v| \end{cases}$$

References

[1] https://algorithmica.org/en/eytzinger,
accessed: December 26th, 2020 16:14

[2] https://sahandsaba.com/understanding-sat-by-implementing-a-simple-sat-solver-in-python.html,
accessed: December 18th, 2020 16:54

[3] Donald E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Addison Wesley, 3rd edition, 1998.

[4] https://en.wikipedia.org/wiki/Coprime_integers,
accessed: December 18th, 2020 12:20

[5] https://en.wikipedia.org/wiki/Heap_(data_structure),
accessed: December 18th, 2020 15:31

[6] https://www.geeksforgeeks.org/array-representation-of-binary-heap/,
accessed: December 18th, 2020 15:40

[7] John Croucher, 2006, Collecting Coupon-A Mathematical Approach, Australian Senior Mathematics Journal https://files.eric.ed.gov/fulltext/EJ744035.pdf,
accessed: December 27th, 2020 14:07

[8] https://en.wikipedia.org/wiki/Coupon_collector,
accessed: December 27th, 2020 14:15

[9] https://tzkt.io/cycles
accessed: December 28th, 2020 11:54

[10] https://bakendorse.com/#/cycles/311/projected_stats,
accessed: December 28th, 2020 12:17

[11] https://en.wikipedia.org/wiki/Bernoulli%27s_inequality#Related_inequalities,
accessed: January 5th, 2021 19:29

[12] Schindelhauer, Christian. *"Chapter 3: First Structures, Part 2: CAN - The Next Level"*, Peer-to-Peer Networks, 2 April 2019, University of Freiburg. Lecture.