Introduction to Horus — Part 2

Formal verification of an automated market maker (AMM)

Julian Sutherland

Follow

Nethermind.eth

--

Listen

Share

By Nethermind's Formal Verification Team. Thanks to

[Julian Sutherland

](https://twitter.com/JulekSU)for reviewing and editing.

An automated market maker is a program (a smart contract) that sets the price of a collection of cryptocurrency tokens. Investors give their assets to the contract, which are held in a "pool". When a user wishes to make a trade, the contract (1) computes an exchange rate, (2) receives the payment tokens (which are added to the appropriate pool), and (3) pays out the desired tokens (from a different pool). In exchange for keeping the pools sufficiently full, investors earn a bit of extra money derived from the transaction fees collected by the contract.

In this blog post, we attempt formal verification of a simple Starknet automated market maker (AMM) using Horus, an open-source CLI tool that uses SMT (satisfiability modulo theory) solvers to check user-defined program properties. The example contract is available on Github in a file called toy_amm_split.cairo

. A working familiarity with the Starknet ecosystem is assumed in what follows, as well as having read the first Horus blog post.

We start by challenging an excerpt that suggests formal verification is of no use. Consider the following subset of the toy_amm_split.cairo

contract:

The function in question is get_pool_token_balance()

. It is only a single line of code that reads from a storage variable called pool_balance()

. This storage variable can be thought of as a global mapping from tokens to balances which can be read from and updated elsewhere in the contract.

In this example, the specification asserts that the return value of the function, denoted $Return.balance

(read: the element of the return tuple of get_pool_token_balance()

named balance

), is equal to the storage variable pool_balance()

indexed at the value of the token_type

argument. Essentially, it is a reimplementation of the function's logic. One could easily argue that if you made a mistake implementing the function itself, you are likely to make the same mistake implementing the specification — this is what we call the set of Horus annotations for a given function. This is known as the [test oracle problem

](https://ieeexplore.ieee.org/document/7422146) in the literature and, as its name suggests, rears its head in both software testing and formal verification.

We can come up with a modified version of our example that demonstrates this danger. Suppose, instead, we have two arguments to get_pool_token_balance()

: one is a counter of some sort, which we increment within the function body, and the other takes the place of token_type: felt

. So we have:

There is now a main()

function, and we call get_pool_token_balance(token_type, counter)

. Note that the first parameter, x

, is the one that gets incremented in get_pool_token_balance()

, and we use the counter as a token_type

when reading from the storage variable! So main()

will certainly not work as we expect.

Now we may run this through Horus and make sense of the output. We see:

So Horus cannot detect the bug. We've mixed up x

and y

in the function body, and we've mixed it up again in the @post

conditions, and thus we have hit the test oracle problem

. One could argue that the bug is really in main()

and we've passed the wrong arguments to get_pool_token_balance()

, but if we had many other functions in our program where the convention is always to pass the counter

as the last argument, it would certainly be the case that the fault is in get_pool_token_balance()

. Name mix-ups where the swapped variables have the same type are one of the hardest sorts of bugs to catch, and are made even more subtle when we choose bad, nondescriptive variable names, as in this version of get_pool_token_balance()

.

We illustrate this shortcoming of formal verification methods (and software testing) to clarify the distinction between cases where using formal tools like Horus is a waste of time and cases where they are nontrivially useful. One might come away from this discussion so far with the opinion that any sort of program verification is futile, so let us look at an example of the latter case. Consider this modified version of our get_pool_token_balance()

function:

Notice that the specification for our function is functionally the same as our first example: @post $Return.balance == pool_balance(h)

. It is just an assertion that we return the pool_balance

at one of the function's arguments. But the logic by which we return this value is (contrivedly) opaque, and may harbor bugs. If we imagine a situation where we perform several actions, make use of intermediate results, and return multiple values, the complexity may be infeasible to clear away (via refactoring the code) while preserving the semantics of the function. This is where formal verification shines. It is able to abstract away a high-complexity implementation in order to guarantee the correctness of a low-complexity specification.

Phrased as a rule of thumb, techniques like software testing and formal verification, in which you write and check assertions about your program, are most appropriate when the top-level assertions are simpler than the code. Here, when we say "simpler", we mean easier for the programmer to understand since a bug in one's assertions renders verification methods useless. The probability of a bug in the assertions must be substantially lower than the probability of a bug in the implementation.

The function above is an extreme example of this: the postcondition is transparent, and the implementation is so unreadable as to be obfuscated.

And indeed, Horus can make sense of this trivially (and verify it!) where it would take a programmer some effort:

Now, let us look at a more substantial portion of our automated market maker contract.

N.B.: We have split up the functionality of the swap function considerably for performance reasons. Horus works best when it is able to leverage function abstraction (replacing a function's instructions with its specification).

Consider the following snippet:

One may reasonably make the objection that the specification for this function (the set of all comments which start with //@

) appears not to satisfy the property we described above: it is, if not complicated, then certainly verbose. This is fair since we assume several data invariants that cannot be expressed within Cairo.

The following postcondition defines the key invariant of this type of automated market maker (known as a constant function, and in particular, a constant product

market maker):

Variables beginning with the $

character are logical variables, defined (and given a type) using a @declare

annotation. These are simply variables that are local to the specification.

Note that from

and to

represent the token the user is converting their money from and the token the user is converting their money to, respectively. The usefulness of this invariant is best observed if we make the simplifying assumption that the amount_from

is 1

. This is like asking, "How many euros can I get for 1 dollar?" We can also safely ignore the remainder term $Return.r

for the moment, since our goal is simply to get an intuitive sense for what the invariant means. Then, renaming our variables to suggest that we're asking for euros equivalent to 1 dollar, the equation becomes

or, equivalently,

It is now possible to get a feel for what happens when the contract starts to run out of a particular token (i.e., when the pool balance approaches zero). As the pool runs out of euros, the denominator shrinks, and we get less and less for our single dollar. Conversely, as the pool runs out of dollars, our dollar becomes more and more valuable since now the denominator is shrinking. In this way, the contract incentivizes users and investors to behave in such a way that it maintains an ample supply of each.

All this to say that this postcondition in our specification is a natural expression of the invariant (we have a product on both sides of our constant product equation).

But the real power of a tool like Horus is its ability to compose function specifications. When we call do_swap_lets()

, we can forget entirely about its implementation and reason solely from its specification. If the preconditions are satisfied when the function is called, then the postcondition is guaranteed. Thus we can add more complexity to the implementation of a function that calls do_swap_lets()

and still obtain the nice postcondition, as seen below:

Notice that our preconditions for do_swap()

look very similar to do_swap_lets()

, which is necessary since we call the latter in the former, and we can write the same postcondition, regardless of what else goes on in the calling function (in particular, the calls to do_swap_from_balance()

and do_swap_to_balance()

), because we are simply returning the corresponding value from do_swap_lets()

, and in verifying do_swap()

we may assume we already have a Verified

spec for that.

One can easily imagine a more complicated call graph, where intermediate results are pulled from many different functions, and in each, the values are transformed slightly. All this is run from a single entry point. Suppose we know the relationship between the inputs and the outputs of the main entry point function. If we are computing roots of some polynomial, then we know the polynomial ought to evaluate to zero at each of the found roots. However, the actual calculation may be quite complicated, involving a large call graph of functions. What Horus gives us is the secure knowledge that the combination of all the (possibly complicated) behaviors of the functions with a relatively larger distance from the entry point does, in fact, guarantee the desired top-level property. If we imagine the simple case where our call graph is a tree, we can write

specifications for all our leaf nodes and the root (entry point) and then let Horus handle the rest, which it may achieve via inlining.

We conclude with a quick discussion of storage variables, which we can specify in the Horus annotation language using the @storage_update

keyword. We can see an example of this in the modify_account_balance()

helper function from the AMM contract, reproduced below:

The annotation is

and quite transparently states that we are adding amount

to the existing storage variable value. An unannotated storage update will result in a False

judgment. N.B.: Whenever you reference a storage variable in an annotation, you are referencing its value before

the function executes. In a @storage_update

annotation, the right-hand side (after the :=

) is the value we are assigning to the storage variable, e.g., the value we assert it has when the function returns.

Try it out yourself! You can find the example AMM contract on Github in a file called[oy_amm_split.cairo]

. Note that that you will have to use the MathSAT5 solver for Horus to verify this file.

The Horus alpha release supports Cairo 0.10.1, with a Cairo 1.0 release coming soon.

New to Starknet and looking for more developer tools? Check out Nethermind'sWarp — the Solidity to Cairo compiler,Juno — Starknet client implementation, and Voyager, a StarkNet block explorer!

About Us

Nethermind is a team of world-class builders and researchers. We empower enterprises and developers worldwide to access and build upon the decentralized web. Our work touches every part of the Web3 ecosystem, from our Nethermind node to fundamental cryptography research and application-layer protocol development.

Disclaimer

Horus is currently in the alpha stage and no guarantee is being given as to the accuracy and/or completeness of any of the outputs the tool may generate. The tool is provided on an 'as is' basis, without warranties or conditions of any kind, either express or implied, including without limitation as to the outputs of the verification process and the security of any system verified using Horus. As per the relevant licenses, to the fullest extent permitted by the law, Nethermind disclaims any liability in connection with your use of Horus and/or any of its outputs.

Please also note that the terminology used by Horus and in the above blog post, including but not limited to words such as 'guarantee', should be interpreted strictly within the remit of formal verification terminology. These words are not intended to, and shall not be construed as having legal significance of any kind.

This blog post has been prepared for the general information and understanding of the readers. No representation or warranty, express or implied, is given by Nethermind as to the accuracy or completeness of the information or opinions contained in the above post. For more information, refer to the ReadMe section of the Horus GitHub page.