Smart contract debugger

Abstract

The aim of this text is to provide a description of general approaches for simplifying debugging of programs written for the TVM (virtual machine used to execute smart contracts in the (Free) TON Blockchain)

Introduction

Being well thought out, in essence a description of TVM given in tvm.pdf by Nikolai Durov it already contains all tools for low-level debugging and the only purpose of this document is to provide ideas how to fit those low-level tools for comfy high-level languages.

TVM is deterministic virtual machine which can be described as `step function f : VmState → VmState`. Any VmState itself is entirely determined by SCCCGL: Stack, Control registers, Current continuation, Current codepage, Gas limits and Dictionary of Libraries.

That means that at any step the execution of smart contract can be stopped, analyzed and continued. The question is how show this information to developer in most convenient way. This document propose the following suggestions described below in separate sections:

- initial state hijacking
- using DEBUG op-codes for state identification
- high-level language <-> VMState conversion and patching
- inter-contract debugging scripts

State hijacking

Since smart-contracts operate in the environment determined by history of joint action of blockchain users it is primary important to recreate this environment as close as possible. In particular it will be handy to have tools that allow to save to file vm registers and VmLibraries corresponding to any particular existed block (on mainnet or testnet). It is also necessary to be able to restore and save contract storage for any contract for any particular block. All those file will be useful for testing in inter-contract debugging scripts.

Using DEBUG op-codes for state identification

For easy mapping of high-level code and TVM execution path we propose to introduce new debug op-code:

`FEF302XXXXXX` - in non-debug regime it works like 6-byte NOP opcode (in accordance with existing vm rules); in debug mode this code initiate serializing VMState (see Appendix 1 for serialization scheme) with corresponding identifier `XXXXXX`.

That way high-level compiler (in debug mode) at the end of each statement should include `FEF302XXXXXX` with unique identifier (up to 16e6 different identifiers are more than enough for any vm run). Simultaneously, compiler should store expected stack content (with binding those stack content to high-level variable names).

After running virtual machine in debug mode one will be able to analyze (see below) state of variables at any logical point of the code.

High-level language <-> VMState conversion and patching

At any point of the code debugger should be able to open corresponding vmstate file and compiler meta-information file and display them. That includes parsing all tvm datatypes (with special attention to slices, cells and dictionaries) from serialized form.

For sophisticated users it will be handy to be able to hot-patch state. In particular, for instance, if developer believe that there should be number `5` on the top of the stack, but there is number `2`, it will be good to change top stack element and continue running VM (since VM next state only depends only on previous one it is, indeed, very simple: debugger just need serialize state back and run vm with it, even identification marks will work). This way developer will be able to identify many bugs at once.

Inter-contract debugging scripts

Since the main idea of TON is asynchronous operation of ensemble of smartontracts, it is very important to test the inter-contract communication.

Linearization Note, while asynchronous nature of contract execution allows simultaneous execution of a set of actions, those simultaneous actions can only be fully independent. That means that the fact that they are simultaneous doesn't matter, which in turn means that they can be replaced by sequence of non-simultaneous actions. That means that developer effectively may write linear (consecutive) script of one event after another without narrowing test scope.

The general form for such scripts may look the following way:

```
let contractState = hijacker.copyState("mainnet",
"0:811ea85643a12bdedf77339ba31fbc5242e268968771abbbab7a61ebbc675db0");
contractState.code = compileCode("contract.sol");
let contract1 = new ContractArtifact(
                  {
                    state: contractState
                    balance: 1 ton,
                    address :
"0:1111111143a12bdedf77339ba31fbc5242e268968771abbbab7a61ebbc675db0"
                  });


let contract2 = contract1.clone();

let message = generateInternalMessageCallFromABI("RootTokenContract.abi.json",
"getWalletCode", {from:contract2.address, value:4 ton});

let outActions = contract1.accept(message);
let response = outActions[0];
require(response.type == InternalMessage);
```

```
contract2.accept(response);
```

As we can see in this script we copy state from mainnet account. Also we load new code which we want to test. Then we generate contract artifact by providing contract state, balance and address (note that address is not part of the state). Then we clone this contract and emulate calling of contract1 and passing internal message generated by transaction to contract2. Note that it will be handy to have functions like `generateInternalMessageCallFromABI` which can generate messages suitable for triggering specific contract methods by ABI. An alternative would be building message from scratch (which is still valuable for testing purposes).

Running this script will generate consequence of inter-contract script statement as well TVM transaction. Due to DEBUG op-codes described above those transactions will be parsable as well, thus allowing precise analysis and control of execution. At the GUI level result of script execution can be represented as table where each line can be associate either with inter-contract debugging script statement or smartcontract code statement, by clicking on any line developer will be able to check local variable. If necessary it will be possible to change local variables and recalculate all next lines.

Note, that even without transaction introspection Inter-contract debugging scripts will be usable for high-level integration tests (it is quite close to Truffle-like tests). Thus it will be good to implement such system as library for popular programming language (like JS), rather than new DSL, that way already existed code can be used for testing real world usecases.

Summary

Deterministic and isolated nature of TVM, as well as linear (or equivalent to linear) path of execution, make it incredibly easy to build debug tools. Given that ideas described in that document are implemented, the main difficulty will be handy GUI.

Appendix 1

Serialization scheme:

TVM stack values. TVM stack values can be serialized as follows:

```
vm_stk_tinyint#01 value:int64 = VmStackValue;
vm_stk_int#0201_ value:int257 = VmStackValue;
vm_stk_nan#02FF = VmStackValue;
vm_stk_cell#03 cell:^Cell = VmStackValue;
_ cell:^Cell st_bits:(## 10) end_bits:(## 10)
{ st_bits <= end_bits }
st_ref:(#<= 4) end_ref:(#<= 4)
{ st_ref <= end_ref } = VmCellSlice;
vm_stk_slice#04 _:VmCellSlice = VmStackValue;
vm_stk_builder#05 cell:^Cell = VmStackValue;
vm_stk_cont#06 cont:VmCont = VmStackValue;
```

TVM stack. The TVM stack can be serialized as follows:

```
vm_stack#_ depth:(## 24) stack:(VmStackList depth) = VmStack;
vm_stk_cons#_ {n:#} head:VmStackValue tail:^(VmStackList n)
= VmStackList (n + 1);
vm_stk_nil#_ = VmStackList 0;
```

TVM control registers. Control registers in TVM can be serialized as follows:

```
_ cregs:(HashmapE 4 VmStackValue) = VmSaveList;
```

TVM gas limits. Gas limits in TVM can be serialized as follows:

```
gas_limits#_ remaining:int64 _:^[
max_limit:int64 cur_limit:int64 credit:int64 ]
= VmGasLimits;
```

TVM library environment. The TVM library environment can be serialized as follows:

```
_ libraries:(HashmapE 256 ^Cell) = VmLibraries;
```

TVM continuations. Continuations in TVM can be serialized as follows:

```
vmc_std$00 nargs:(## 22) stack:(Maybe VmStack) save:VmSaveList
cp:int16 code:VmCellSlice = VmCont;
vmc_envelope$01 nargs:(## 22) stack:(Maybe VmStack)
save:VmSaveList next:^VmCont = VmCont;
vmc_quit$1000 exit_code:int32 = VmCont;
vmc_quit_exc$1001 = VmCont;
vmc_until$1010 body:^VmCont after:^VmCont = VmCont;
vmc_again$1011 body:^VmCont = VmCont;
vmc_while_cond$1100 cond:^VmCont body:^VmCont
after:^VmCont = VmCont;
vmc_while_body$1101 cond:^VmCont body:^VmCont
after:^VmCont = VmCont;
vmc_pushint$1111 value:int32 next:^VmCont = VmCont;
```

TVM state. The total state of TVM can be serialized as follows:

```
vms_init$00 cp:int16 step:int32 gas:GasLimits
stack:(Maybe VmStack) save:VmSaveList code:VmCellSlice
lib:VmLibraries = VmState;
vms_exception$01 cp:int16 step:int32 gas:GasLimits
exc_no:int32 exc_arg:VmStackValue
save:VmSaveList lib:VmLibraries = VmState;
vms_running$10 cp:int16 step:int32 gas:GasLimits stack:VmStack
save:VmSaveList code:VmCellSlice lib:VmLibraries
= VmState;
vms_finished$11 cp:int16 step:int32 gas:GasLimits
exit_code:int32 no_gas:Boolean stack:VmStack
save:VmSaveList lib:VmLibraries = VmState;
```