DSL$_{\text{EVM}}$: Domain-Specific Language for EVM Specifications

Daejun Park, Yi Zhang, and Grigore Rosu

Runtime Verification, Inc.

February 27, 2018

The K-framework provides the reachability logic theorem prover[1] that is parameterized by the langauge semantics. Instantiated with the KEVM[2], a complete formal semantics of the Ethereum Virtual Machine (EVM), the K prover yields a correct-by-construct deductive program verifer for the EVM. The EVM verifier takes an EVM bytecode and a specification as inputs, and automatically proves that the bytecode satisfies the specification, if it is the case. The EVM specification essentially specifies the pre- and post-conditions of the EVM bytecode in the form of the reachability logic claims.

We present a domain-specific language (DSL) for EVM specifications, called DSL$_{\text{EVM}}$, to succintly specify the specifications. DSL$_{\text{EVM}}$ consists of two parts: high-level notations, and specification templates and template parameters.

The DSL$_{\text{EVM}}$ high-level notations make EVM specifications more succinct and closer to their high-level specifications. The succinctness increases the readability, and the closeness helps "eye-ball validation" of the specification refinement. The high-level notations are defined by translation to the corresponding EVM terms, and thus can be freely used with other EVM terms. The notations are inspired by the production compilers of the smart contract languages like Solidity and Viper, and their definition is derived by formalizing the corresponding translation made by the compilers.

Even with the high-level notations, the refined EVM specification is still quite large due to the sheer size of the KEVM configuration, while large part of that is the same across the different specifications. The specification template allows to reuse and share the common part over the different specifications, avoiding duplication. The template is essentially a specification but contains several parameters which are place-holders to be filled with parameter values when being instantiated. The template is supposed to be instantiated with the parameter values for each specification.

1 DSL$_{\text{EVM}}$ Notations

ABI Call Data When a function is called in EVM, its arguments are encoded in a single byte-array and put in the so-called 'call data' section. The encoding is defined in the Ethereum contract application binary interface (ABI) specification[3]. DSL$_{\text{EVM}}$ provides `#abiCallData`, a notation to specify the ABI call data in a way similar to a high-level function call notation, defined in Figure 1. It specifies the function name and the (symbolic) arguments along with their types. For example, the following abstraction represents a data that encodes a call to the `transfer` function with two arguments: `TO`, the receiver account address of type `address` (an 160-bit unsigned integer), and `VALUE`, the value to transfer of type `unit256` (a 256-bit unsigned integer).

```
#abiCallData("transfer", #address(TO), #uint256(VALUE))
```

which denotes (indeed, is translated to) the following byte array:

[1]`http://fsl.cs.illinois.edu/index.php/Semantics-Based_Program_Verifiers_for_All_Languages`
[2]`https://github.com/kframework/evm-semantics`
[3]`https://solidity.readthedocs.io/en/develop/abi-spec.html`

```
syntax TypedArg ::= "#uint160"      "(" Int ")"
                  | "#address"      "(" Int ")"
                  | "#uint256"      "(" Int ")"

syntax TypedArgs ::= List{TypedArg, ","}                [klabel(typedArgs)]

syntax WordStack ::= #abiCallData( String , TypedArgs ) [function]
rule #abiCallData( FNAME , ARGS )
  => #parseByteStack(substrString(Keccak256(#generateSignature(FNAME, ARGS)), 0, 8))
  ++ #encodeArgs(ARGS)

syntax String ::= #generateSignature        ( String, TypedArgs)    [function]
                | #generateSignatureAux     ( String, TypedArgs)    [function]
rule #generateSignature( FNAME , ARGS ) => #generateSignatureAux(FNAME +String "(", ARGS)
//
rule #generateSignatureAux(SIGN, TARGA, TARGB, TARGS)
  => #generateSignatureAux(SIGN +String #typeName(TARGA) +String ",", TARGB, TARGS)
rule #generateSignatureAux(SIGN, TARG, .TypedArgs)
  => #generateSignatureAux(SIGN +String #typeName(TARG), .TypedArgs)
rule #generateSignatureAux(SIGN, .TypedArgs) => SIGN +String ")"

syntax String ::= #typeName ( TypedArg ) [function]
rule #typeName(#uint160( _ ))  => "uint160"
rule #typeName(#address( _ ))  => "address"
rule #typeName(#uint256( _ ))  => "uint256"

syntax WordStack ::= "#encodeArgs" "(" TypedArgs ")" [function]
rule #encodeArgs(ARG, ARGS)    => #getData(ARG) ++ #encodeArgs(ARGS)
rule #encodeArgs(.TypedArgs)   => .WordStack

syntax WordStack ::= "#getData" "(" TypedArg ")" [function]
rule #getData(#uint160( DATA )) => #asByteStackInWidth( DATA , 32 )
rule #getData(#address( DATA )) => #asByteStackInWidth( DATA , 32 )
rule #getData(#uint256( DATA )) => #asByteStackInWidth( DATA , 32 )
```

Figure 1: Definition of `#abiCallData`

```
F1 : F2 : F3 : F4 : T1 : ... : T32 : V1 : ... : V32
```

where `F1 : F2 : F3 : F4` is the (two's complement) byte-array representation of `2835717307`, the hash value of the `transfer` function signature ABI encoding, "`keccak256("transfer(address,unit256)")`", and `T1 : ... : T32` and `V1 : ... : V32` are the byte-array representations of `TO` and `VALUE` respectively.

ABI Event Log EVM logs are special data structures in the blockchain, being searchable by off-chain clients. Events are high-level wrappers of the EVM logs provided in the high-level languages. Contracts can declare and generate the events, which will be compiled down to EVM bytecode using the EVM log instructions. The encoding scheme of the events in the EVM logs is defined in the Ethereum contract application binary interface (ABI) specification, leveraging the ABI call data encoding scheme.

DSL_{EVM} provides `#abiEventLog`, a notation to specify the EVM logs in the high-level events, defined in Figure 2. It specifies the contract account address, the event name, and the event arguments. For example, the following notation represents an EVM log data that encodes the `Transfer` event generated by the `transfer` function, where `ACCT_ID` is the account address, and `CALLER_ID`, `TO_ID`, and `VALUE` are the event arguments. Each argument is tagged with its ABI type (`#address` or `#uint256`), and the `indexed` attribute (`#indexed`) if

```
syntax EventArgs ::= List{EventArg, ","} [klabel(eventArgs)]
syntax EventArg  ::= #indexed(TypedArg)
                   | TypedArg

syntax SubstateLogEntry ::= #abiEventLog(Int, String, EventArgs) [function]
rule #abiEventLog(ACCT_ID, EVENT_NAME, EVENT_ARGS)
  => { ACCT_ID | #getEventTopics(EVENT_NAME, EVENT_ARGS) | #getEventData(EVENT_ARGS) }

syntax WordStack ::= #getEventTopics(String, EventArgs) [function]
rule #getEventTopics(ENAME, EARGS)
  => #parseHexWord(Keccak256(#generateSignature(ENAME, #getTypedArgs(EARGS))))
   : #getIndexedArgs(EARGS)

syntax TypedArgs ::= #getTypedArgs(EventArgs) [function]
rule #getTypedArgs(#indexed(E), ES) => E, #getTypedArgs(ES)
rule #getTypedArgs(E:TypedArg,  ES) => E, #getTypedArgs(ES)
rule #getTypedArgs(.EventArgs)      => .TypedArgs

syntax WordStack ::= #getIndexedArgs(EventArgs) [function]
rule #getIndexedArgs(#indexed(E), ES) => #getValue(E) : #getIndexedArgs(ES)
rule #getIndexedArgs(_:TypedArg,  ES) =>                #getIndexedArgs(ES)
rule #getIndexedArgs(.EventArgs)      => .WordStack

syntax WordStack ::= #getEventData(EventArgs) [function]
rule #getEventData(#indexed(_), ES) =>                #getEventData(ES)
rule #getEventData(E:TypedArg,  ES) => #getData(E) ++ #getEventData(ES)
rule #getEventData(.EventArgs)      => .WordStack

syntax Int ::= "#getValue" "(" TypedArg ")" [function]
rule #getValue(#uint160(V)) => V
rule #getValue(#address(V)) => V
rule #getValue(#uint256(V)) => V
```

Figure 2: Definition of #abiEventLog

any, according to the event declaration in the contract.

```
#abiEventLog(ACCT_ID, "Transfer",
      #indexed(#address(CALLER_ID)), #indexed(#address(TO_ID)), #uint256(VALUE))
```

The above notation denotes (i.e., is translated to) the following EVM log data structure:

```
{ ACCT_ID
| 100389287136786176327247604509743168900146139575972864366142685224231313322991
: CALLER_ID
: TO_ID
: .WordStack
| #asByteStackInWidth(VALUE, 32)
}
```

where "100389287136786176327247604509743168900146139575972864366142685224231313322991" is the hash value of the event signature, "`keccak256("Transfer(address,address,unit256)")`".

Hashed Location for Storage The storage accommodates permanent data such as the `balances` map. A map is laid out in the storage where the map entries are scattered over the entire storage space for which the

```
syntax IntList ::= List{Int, ""} [klabel(intList)]

syntax Int ::= #hashedLocation(String, Int, IntList) [function]
rule #hashedLocation(LANG, BASE, .IntList) => BASE
//
rule #hashedLocation("Viper",    BASE, OFFSET OFFSETS)
  => #hashedLocation("Viper",    keccakIntList(BASE) +Word OFFSET, OFFSETS)
rule #hashedLocation("Solidity", BASE, OFFSET OFFSETS)
  => #hashedLocation("Solidity", keccakIntList(OFFSET BASE),       OFFSETS)

syntax Int ::= keccakIntList(IntList) [function]
rule keccakIntList(VS) => keccak(intList2ByteStack(VS))

syntax WordStack ::= intList2ByteStack(IntList) [function]
rule intList2ByteStack(.IntList) => .WordStack
rule intList2ByteStack(V VS) => #asByteStackInWidth(V, 32) ++ intList2ByteStack(VS)
  requires 0 <=Int V andBool V <Int (2 ^Int 256)
```

Figure 3: Definition of `#hashedLocation`

(256-bit) hash of each key is used to determine the location. The detailed mechanism of calculating the location varies by compilers. In Viper, for example, `map[key1][key2]` is stored at the location:

```
hash(hash(idx(map)) + key1) + key2
```

where `idx(map)` is the position index of `map` in the program, and `+` is the addition modulo 2^{256}, while in Solidity, it is stored at:

```
hash(key2 ++ hash(key1 ++ idx(map)))
```

where `++` is the byte-array concatenation.

DSL$_{\text{EVM}}$ provides `#hashedLocation` that allows to uniformly specify the locations in a form parameterized by the underlying compilers. For example, the location of `map[key1][key2]` can be specified as follows, where `{COMPILER}` is a place-holder to be replaced by the name of the compiler. Note that the keys are separated by the white spaces instead of commas.

```
#hashedLocation({COMPILER}, idx(map), key1 key2)
```

This notation makes the specification independent of the underlying compilers, enabling it to be reused for differently compiled programs. Specifically, `#hashedLocation` is defined as in Figure 3, capturing the storage layout schemes of Solidity and Viper.

2 Specification Template

EVM specifications are written over the full KEVM configuration. However, large part of the configuration is not relevant for functional correctness specification and can be shared across the different specifications. DSL$_{\text{EVM}}$ allows to specify a template specification that can be instantiated for each specification.

Figure 4 shows the template specification. Essentially, it is a reachability claim over the KEVM configurations. The configurations are generalized as much as possible to capture arbitrary contexts in which contract functions are called. The underline (`_`) is an anonymous/nameless variable (with no constraint) that matches any value, denoting an arbitrary state. The constant values and the `requires` condition represent the pre-condition. The upper-case variables enclosed by the curly braces (`{`, `}`) are the template parameters, i.e., place-holders to be replaced with parameter values when being instantiated. Note that there are only a few of parameters, meaning that the large part of the specification is shared across the different specifications.

```
rule
  <k> {K} </k>
  <exit-code> 1 </exit-code>
  <mode> NORMAL </mode>
  <schedule> BYZANTIUM </schedule>
  <ethereum>
    <evm>
      <output> {OUTPUT} </output>
      <memoryUsed> {MEMORYUSED} </memoryUsed>
      <callDepth> CALL_DEPTH </callDepth>
      <callStack> _ => _ </callStack>
      <interimStates> _ </interimStates>
      <substateStack> _ </substateStack>
      <callLog> .Set </callLog> // for vmtest only
      <txExecState>
        <program> #asMapOpCodes(#dasmOpCodes(#parseByteStack({CODE}), BYZANTIUM)) </program>
        <programBytes> #parseByteStack({CODE}) </programBytes>
        <id> ACCT_ID </id>
        <caller> CALLER_ID </caller> // msg.sender               |    <previousHash> _ </previousHash>
        <callData> {CALLDATA} </callData> // msg.data            |    <ommersHash> _ </ommersHash>
        <callValue> {CALLVALUE} </callValue> // msg.value        |    <coinbase> _ </coinbase>
        <wordStack> {WORDSTACK} </wordStack>                     |    <stateRoot> _ </stateRoot>
        <localMem> {LOCALMEM} </localMem>                        |    <transactionsRoot> _ </transactionsRoot>
        <pc> {PC} </pc>                                          |    <receiptsRoot> _ </receiptsRoot>
        <gas> {GAS} </gas>                                       |    <logsBloom> _ </logsBloom>
        <previousGas> _ => _ </previousGas>                      |    <difficulty> _ </difficulty>
        <static> false </static> // NOTE: non-static call        |    <number> _ </number>
      </txExecState>                                             |    <gasLimit> _ </gasLimit>
      <substate>                                                 |    <gasUsed> _ </gasUsed>
        <selfDestruct> _ </selfDestruct>                         |    <timestamp> NOW </timestamp> // now
        <log> {LOG} </log>                                       |    <extraData> _ </extraData>
        <refund> {REFUND} </refund>                              |    <mixHash> _ </mixHash>
      </substate>                                                |    <blockNonce> _ </blockNonce>
      <gasPrice> _ </gasPrice>                                   |    <ommerBlockHeaders> _ </ommerBlockHeaders>
      <origin> ORIGIN_ID </origin> // tx.origin                  |    <blockhash> _ </blockhash>
    </evm>
    <network>
      <activeAccounts> ACCT_ID |-> false _:Map </activeAccounts>
      <accounts>
        <account>
          <acctID> ACCT_ID </acctID>
          <balance> _ </balance>
          <code> #parseByteStack({CODE}) </code>
          <storage> {STORAGE} </storage>
          <nonce> _ </nonce>
        </account>
     // ... // TODO: fix
      </accounts>
      <txOrder> _ </txOrder>
      <txPending> _ </txPending>
      <messages> _ </messages>
    </network>
  </ethereum>
  requires 0 <=Int ACCT_ID    andBool ACCT_ID    <Int (2 ^Int 160)
   andBool 0 <=Int CALLER_ID  andBool CALLER_ID  <Int (2 ^Int 160)
   andBool 0 <=Int ORIGIN_ID  andBool ORIGIN_ID  <Int (2 ^Int 160)
   andBool 0 <=Int NOW        andBool NOW        <Int (2 ^Int 256)
   andBool 0 <=Int CALL_DEPTH andBool CALL_DEPTH <Int 1024
   {REQUIRES}
```

Figure 4: DSL_{EVM} Specification Template

2.1 Template Parameters

The specification parameters and their values are given in the INI format[4], which is essentially a list of the parameter name and value pairs. More precisely, they are given in a variant of the INI format, extended with support for nested inheritance, allowing further reusing parameters over the different but similar specifications. Moreover, the specification parameters are grouped into two categories: function-specific parameters and program-specific parameters. The program-specific parameters are shared among the specifications of the same program. An EVM specification can be represented in terms of the parameter values, using which the full specification is be derived from the specification template.

Function-Specific Parameters Below is an example of the function-specific parameter definition of a `balanceOf` function which takes as an input the account address and returns its balance if the address is non-zero, otherwise throws.

```
[DEFAULT]
output: _
memoryUsed: 0 => _
callValue: 0
wordStack: .WordStack => _
localMem: .Map => _
pc: 0 => _
gas: {GASCAP} => _
log: _
refund: _

[balanceOf]
callData: #abiCallData("balanceOf", #address(OWNER))
storage: #hashedLocation({COMPILER}, {_BALANCES}, OWNER) |-> VALUE _:Map
requires: andBool 0 <=Int OWNER andBool OWNER <Int (2 ^Int 160)
          andBool 0 <=Int VALUE andBool VALUE <Int (2 ^Int 256)

[balanceOf-success]
k: #execute => (RETURN RET_ADDR:Int 32 ~> _)
localMem: .Map => .Map[ RET_ADDR := #asByteStackInWidth(VALUE, 32) ] _:Map
+requires: andBool OWNER =/=Int 0

[balanceOf-failure]
k: #execute => #exception
+requires: andBool OWNER ==Int 0
```

- The parameter definition consists of lists of parameter and value pairs, called 'sections'. Each section begins with its name, enclosed by the square brackets, e.g., `[DEFAULT]` or `[balanceOf]`. The section name is referred to when being inherited by other sections. Each parameter is defined by a key and value pair separated by the colon (`:`).
- A section can inherit another section to avoid duplication and highlight differences between different but similar specifications (sections). For example, `[balanceOf-success]` inherits `[balanceOf]` which in turns inherits `[DEFAULT]`. When a child section inherits its parent section, it overwrites the parameter values except for the parameter key whose name starts with the `+` symbol, for which it accumulates

[4] `https://en.wikipedia.org/wiki/INI_file`

the values by appending the child values to the parent ones. For example, the final value of `requires` of `[balanceOf-success]` is the concatenation of `[DEFAULT]`'s `requires`, `[balanceOf]`'s `+requires`, and `[balanceOf-success]`'s `+requires` values.

- The `[DEFAULT]` section is inherited by all other sections. It specifies the default parameter values.
- The anonymous/nameless variable (`_`) denotes an arbitrary context whose value can be arbitrary and are not relevant w.r.t. functional correctness specification. If it appears in the left-hand side of `=>`, it means that all possible input states/values are considered. If it appears in the right-hand side of `=>`, it means that the output states/values may be updated and different from the input states, but their specific contents are not relevant for the current specification. If it appears solely without `=>`, it means that all possible input states/values are considered but they are not updated at all during the execution, must remain intact.
- $\star$`Int` (e.g., `^Int` and `<=Int`) are (mathematical) integer arithmetic operations.
- `output` specifies the output value at the end of the current transaction. Specifically, in the `[DEFAULT]` section, it specifies that the output value is not relevant for the current specification.
- `memoryUsed` specifies the amount of memory used at the beginning and the end of the execution, and `localMem` specifies the pre- and post-states of the local memory. Specifically, in the `[DEFAULT]` section, it specifies that it begins with the empty (fresh) memory but will end with some used memory whose contents are not relevant for the current specification.
- `callValue` specifies the number of Wei sent with the current transaction.
- `wordStack` specifies the pre- and post-states (snapshots) of the local stack. Specifically, in the `[DEFAULT]` section, it specifies that it begins with the empty (fresh) stack but will end with some elements pushed, but the contents are not relevant for the current specification.
- `pc` specifies the program-counter (PC) values at the beginning and the end of the execution. Specifically, in the `[DEFAULT]` section, it specifies that it starts with PC 0 and will end with some PC value that are not relevant for the specification.
- `gas` specifies the maximum gas amount, `{GASCAP}`, another parameter to be given by the program specification, ensuring that the program does not consume more gas than the limit. One can set a tight amount of the gas limit to ensure that the program does not consume more gas than expected (i.e., no gas leakage). In case that there is no loop in the code, however, one can simply give a loose upper-bound. The verifier proves that the gas consumption is less than the provided limit, and also reports the exact amount of gas consumed during the execution. Indeed, it reports a set of the amounts since the gas consumption varies depending on the context (i.e., the input parameter values and the state of the storage).
- `log` specifies the pre- and post-states of log messages generated. Specifically, in the `[DEFAULT]` section, it specifies that that no log is generated during the execution, while the existing log messages are not relevant.
- `refund` specifies the pre- and post-amounts of the gas refund. Specifically, in the `[DEFAULT]` section, it specifies that no gas is refunded. Note that it does not mean it consumes all the provided gas. The gas refund is different from returning the remaining gas after the execution. It is another notion to capture some specific gas refund events that happen, for example, when an unused storage entry is re-claimed (i.e., garbage-collected). The specification ensures that no such event happens during the execution of the current function.
- `callData` specifies the call data using the `#abiCallData` DSL_{EVM} notation. Specifically, in the `[DEFAULT]` section, it specifies the (symbolic) value, `OWNER`, of the first parameter.

- `storage` specifies the pre- and post-states of the permanent storage. Specifically, in the `[DEFAULT]` section, it specifies that the value of `balances[OWNER]` is `VALUE` and other entries are not relevant (and could be arbitrary values). It refers to another two parameters, `{COMPILER}` and `{_BALANCES}`, which are supposed to be given by the program specification. `{COMPILER}` specifies the language in which the program is written. `{_BALANCES}` specifies the position index of `balances` global variable in the program.

  Specifying the irrelevant entries implicitly expresses the non-interference property. That is, the total supply value will be returned regardless of what the other entires of the storage are. This representation of the irrelevant part is used throughout the entire specification, ensuring one of the principal security properties.

- `k` specifies that the execution eventually reaches the `RETURN` instruction, meaning that the program will successfully terminate. The `RETURN` instruction says that a 32-byte return value will be stored in the memory at the location `RET_ADDR`. The followed underline means that there will be more computation tasks to be performed (e.g., cleaning up the VM state) but they are not relevant w.r.t. the correctness.

- `localMem` in the `[balanceOf-success]` section overwrites that of `[DEFAULT]`, specifying that the local memory is empty in the beginning, but in the end, it will store the return value `VALUE` at the location `RET_ADDR` among others. The other entries are represented by the anonymous variable `_`, meaning that they can be arbitrary and are not relevant w.r.t. the correctness.

- `requires` specifies the pre-condition over the named variables. Specifically, `requires` of the `[DEFAULT]` section specifies the range of symbolic values based on their types. `+requires` of `[balanceOf-success]` specifies additional pre-condition on the top of that of `[DEFAULT]`, which is the condition for the success case, while that of `[balanceOf-failure]` specifies additional pre-condition for the failure case.

Program-Specific Parameters Below is an example of the program-specific parameters.

```
[DEFAULT]
compiler: "Solidity"
_balances: 0
_allowances: 1
_totalSupply: 2
code: "0x60606040526004361061008e57600...0029"
gasCap: 100000
```

- `compiler` specifies the language in which the smart contract is implemented.

- `code` is the EVM bytecode generated from the source code.

- The position index parameters (the ones starting with the underline '`_`') specify the indexes of the storage variables in the order that their declarations appear in the source code, meaning that the `balances` variable is declared first, followed by the `allowances` variable being declared, and followed by the `totalSupply` variable at last. Note that the actual variable names in the source code may be different with the parameter names. The literal name of the variable is not relevant for the EVM specification because the variable name disappears during compilation to EVM, being identified only by it position index.

- `gasCap` specifies the gas limit. Here we give a rough upper-bound for demonstration purposes. In practice, one should set a reasonable amount of the gas limit to see if the program does not consume too much gas (i.e., no gas leakage).