EDB: Debugger for Ethereum's Programming Languages

Report #4: System Design
Advised by Dr. Jackowitz
University of Scranton

Andrew Plaza

October 31, 2018

Abstract

1. Hello

Contents

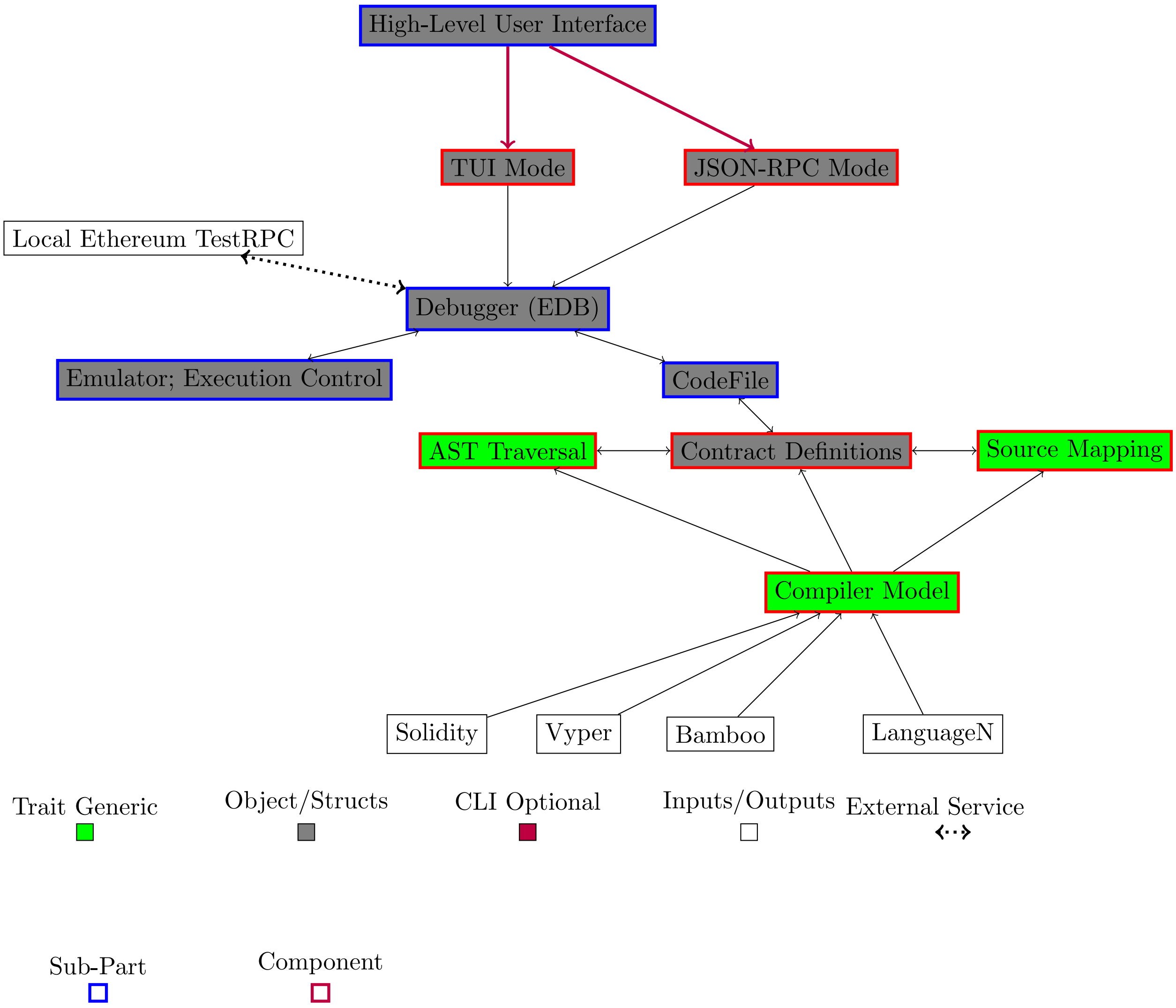
0.1 Introduction

The EDB client application may be launched via the shell with the ‘edb’ command, which launches the main program routine for the binary. Depending upon the options the user has passed EDB, the main program will first connect to the locally-run ethereum test node the user has setup, which EDB references throughout the rest of its execution. The solidity source code file that will be debugged must also be provided by the user. From these inputs EDB creates two of it’s highest-level abstractions, the File Model and the Emulator model, which make up the functionality of the debugger. The file model handles objects created by the compilation of the target source-language, such as the Abstract Syntax Tree, Source Mappings, and Bytecode. The Emulator model handles execution control and stores the Ethereum Virtual Machine(EVM) state at different steps in execution. State that is stored includes the current EVM Stack, Memory, and Non-Volatile Storage. Internally, the Emulator model uses the ‘sputnikvm’ library. These parts make up the core of the debuggers functionality. Built on top of these items is also an optional JSON-RPC, which may be used in order to build featureful Graphical User Interfaces from 3rd-party applications.

The design of EDB started first with a general development plan where models that were needed were identified. Based upon this plan, generic interfaces were created based upon the needs of the structure. Once this was complete, development was constrained to one part or sub-part of one model/interface, which worked towards fitting into the generic interface that was created during the first step. Unit tests were created alongside the original development of the part or sub-part, and testing took place to ensure the part worked individually before moving on to connect the part with the rest of the program. A kanban board on ‘Github Projects‘ was used in order to keep track of work that is under development, finished, or needs to be started.

0.2 Levels

0.2.1 High Level System Model



0.2.2 User Interface Modes

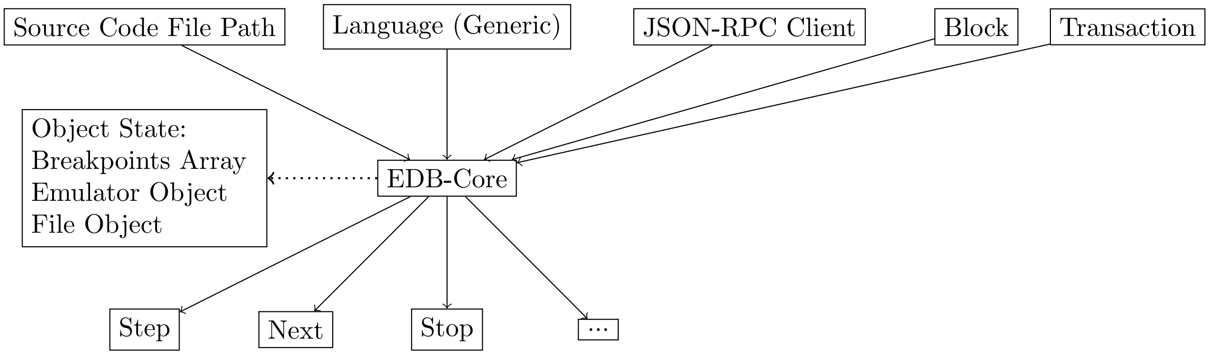
0.2.2.1 Abstract Specification

0.2.2.2 Algorithm Design

0.2.2.3 Component: TUI Mode

0.2.2.4 Component: JSON-RPC Mode

0.2.3 Debugger Core, EDB



0.2.3.1 Abstract Specification

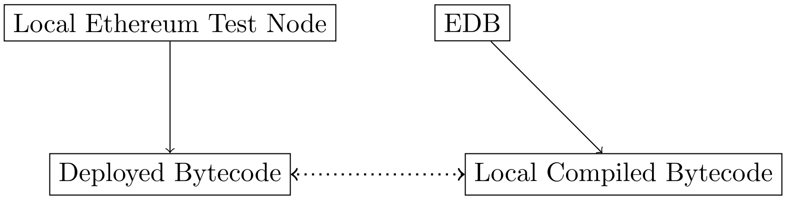
The Debugger core provides all the core debugging features one may expect a debugger to offer. This level is not concerned with parsing input or events that the user may trigger. Instead, it works with input that has already been parsed. Namely, the Ethereum Language Trait, a JSON-RPC Client, and parameters for the transaction the user wishes to debug. Sourcemapping and EVM Execution control is done implicitly through the debug functions: 'step', 'next', 'stop', 'restart', etc. Therefore, this level mostly serves as a higher-level abstraction over the next two levels 'Emulator' and 'CodeFile'. This level accepts the source code file, a object which implements the trait 'Language', a JSON-RPC client to communicate with the locally running Ethereum Test Node, along with the block and transaction the user wishes to debug. From these inputs, creation of the Emulation and Source Mapping objects is enabled.

0.2.3.2 Interface Design

This level should be able to provide all the debug functions needed in order to debug a source code file. It should not expose any objects or functions which deal with Source Mapping or Ethereum Virtual Machine emulation. It is assumed that inputs provided to this interface match up with information that is present in the local test ethereum node. Necessarily this indicates that the compiled bytecode of the source code provided matches with bytecode that has already been deployed onto the Ethereum Test Node. This level should provide a programmatic view of the functions outlined in the requirements specification, namely:

- **step**: step forward to the next source-line in execution
- **stepback**: step back one source-line in execution
- **print**: print stack, memory, or variables in the program
- **printline**: print the current line in the source code that execution is at
- **restart**: restart execution entirely from a clean Ethereum Virtual Machine state
- **run**: run a function in the source code (accepts arguments according to function parameters)
- **next**: Continue execution to the next breakpoint
- **previous**: Take execution back to the previous breakpoint
- **stepinto**: Step into a function (if the current line contains a function call)

0.2.3.3 Component: Address Cache

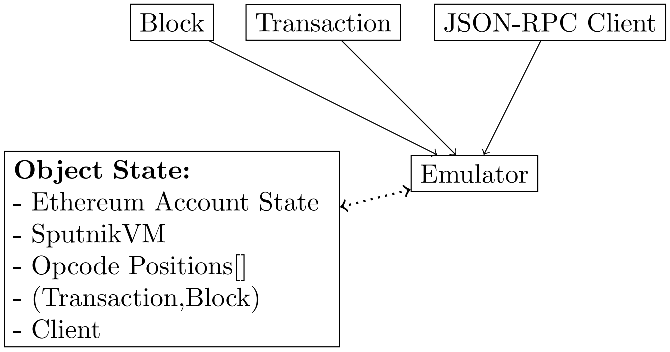


In order to associate information provided by the user in the form of a source code file, a component to match up the compiled bytecode created from the file provided by the user to an address existing in the Ethereum Test Node is necessary. This information is needed throughout transaction execution in case the Ethereum Virtual Machine is in need of any state existing on the blockchain before execution has started. Therefore, a local cache of all known addresses and associated compiled bytecode is created once the debugger is first launched. This cache is stored as a HashMap: The key is a 160bit address (The Address of the Ethereum Account owning the code) and the value is the compiled bytecode. This information is gathered and updated whenever the EDB client is run by crawling all addresses known to the local Ethereum node. It is up to the user whether to refresh this information on each run of the debugger, therefore not storing any persistant state on-disk, or to write this cache to be retrieved on the next run of the debugger. Since only one contract is being debugged at a time, the address cache is only queried once for the relevant address and bytecode. The cache object is not stored along with the Object State, since later retrieval may be done via the filesystem if requested by the user.

0.2.3.4 Component: Breakpoints

Breakpoints are set and unset by two functions 'set_breakpoint' and 'unset_breakpoint'. These are stored in a simple array. Each breakpoint serves as an indicator as to where execution should stop for the rest of the debug functions. For instance, in order to begin debugging the 'set_breakpoint' function of the EDB interface is called first. Once 'run' is called after that, execution is expected to continue up to and only until that line in the original source code file. Each consequent 'step' call is expected to run only until the next line in the source code, while a 'next' call would be expected to run to the next breakpoint.

0.2.4 Emulator, Execution Control



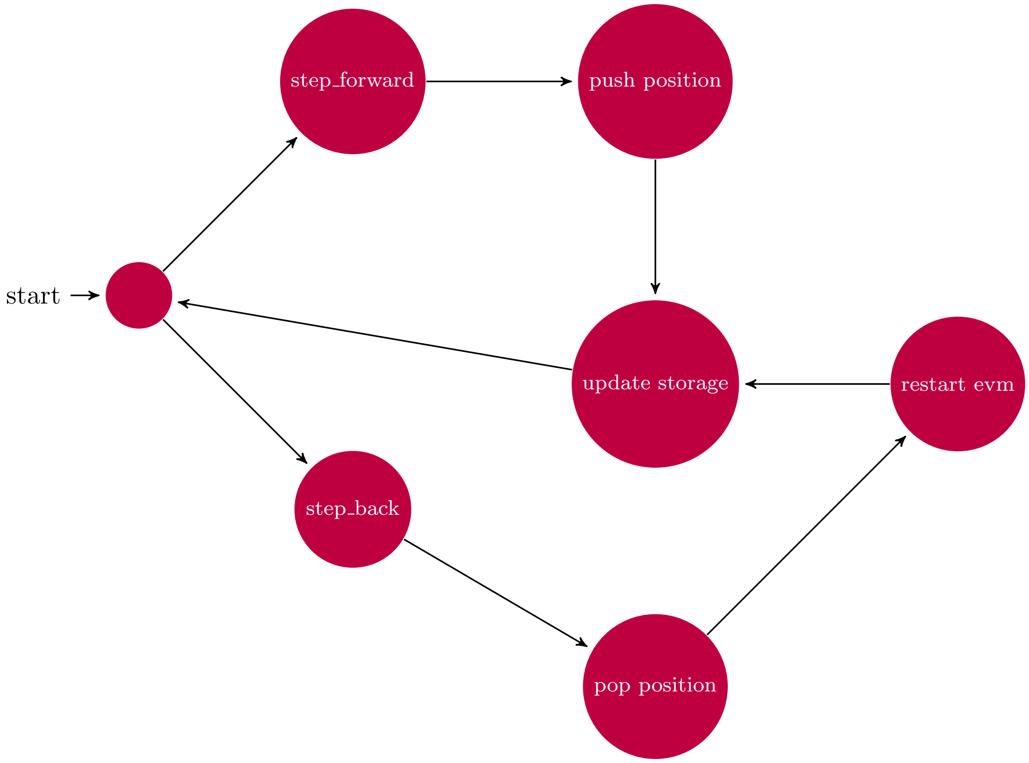
0.2.4.1 Abstract Specification

The Emulator provides execution control for the debugger level. This means it deals at the Opcode and Instruction level, stores and exposes state related to the Ethereum Virtual Machine, and handles any blockchain state requirements it may need from the local ethereum test node. The 'Transaction' and 'Block' inputs mostly contain extraneous information that is unnecessary to debugging but necessary to general Ethereum Transaction Execution, except for the bytecode which is apart of the 'Transaction' input object. The Emulator, however, is not concerned with lines in the source code, breakpoints, or any 'human-decipherable' meaning which may be attached to items that reside in the Ethereum Virtual Machine state. It is able to decipher individual opcodes from Instructions, as well as identify how those opcodes contribute to the next state of the Ethereum Virtual Machine. At it's lowest level, an Ethereum Virtual Machine library, 'sputnikvm' is used in order to carry out these instructions and general code execution. In addition, the Emulator keeps the Stack, Memory, and Non-Volatile storage in sync with the local ethereum test node, and is able to access these data-structures at-will (without any constraints).

0.2.4.2 Interface Design

The interface of the Debugger and Emulator level hold many similiarites. For instance, the Emulator is able to ‘step_forward‘, ‘step_back‘, and ‘run_until‘ a certain point in execution has been reached. However, instead of stepping over lines in source code, functions at the Emulator level deal only with stepping instructions in the bytecode. Therefore, ’step_forward’ means execute the next instruction in the bytecode array, and ’step_back’ means to step back to the overall state before the current instruction had been executed. Functions to peek at the current state of the Virtual Machine are also exposed, in order to faciliate variable-decoding at in the Debugger level.

0.2.4.3 Algorithm Design



There are two main parts of the Emulators Algorithm Design: In-Memory representation of Non-Volatile Storage, and states of the EVM.

Emulator States:

The Emulator is required to have the ability to reference all state of the Ethereum Virtual Machine of executing code during any point in execution. This means that the Stack and Memory for every line in source code must be stored. This is done in order to faciliate stepping through all possible points in execution (including stepping backward). The easiest way of doing so would be to save a copy of the EVM Stack, Memory, and Program Counter in a as a tuple in a dynamically-sized stack. The top of the stack would be the previous state of the Ethereum Virtual Machine. If a user needs to step back, then this state-stack could be popped and the resulting states in the EVM reverted. This, however, may require a large amount of memory depending on the size of the program, and would increase considerably if the Ethereum Smart Contract were to call an external library. This is because an external library requires its own and entirely seperate EVM Stack, Memory (CITE ETHEREUM YELLOW PAPER) (As per the specification in the Ethereum Yellow Paper). The actual execution of smart contracts, however, is inexpensive. Therefore, in order to decrease the memory-footprint of the Emulator, only the opcode-position in the bytecode-array of the corresponding line in source code is stored by the Emulator in a stack. If a user wishes to step back, then, the current EVM state is first totally wiped, the positions-vector popped, and the EVM re-run to the corresponding opcode position that was popped from the positions vector.

Non-Volatile Storage

A user may want to chain a second (or more) transactions after executing one transaction that acts on the state from the first transaction. Therefore, the state of non-volatile storage is kept track of on a per-contract basis that is seperate from the state stored in the Ethereum Test Node. If the user needs to step back, the state changed only from the currently-executing contract is wiped and re-populated along with the EVM state. This state is kept in a nested hashmap, where the first level of the hashmap is a mapping between account addresses (160bit hashes) and a hashmap. The second level is a mapping between unsigned-256bit integers and unsigned-256bit integer. This second level is the 'Non-Volatile' storage for a particular smart contract identified by an account address at the first level.

0.2.4.4 Component: Persistant Account State

In order to faciliate a clean test environment for users wishing to test code, there must be an aspect of seperation from the verbatim Ethereum Test Node State. In other words, the debugger must present an environment where the user may manipulate and mutate their program without having to worry about affecting any change in the node that is running locally, and before committing the changes to the node. This is done in sequentially:

1. Pull the current state from the local Ethereum Test Node before execution of the user's program
2. Handle state updates originating from users program that is being debugged within the debugger
3. Allow user to run other functions/programs/mutate the state further based based on the previously debugged program (chaining execution)
4. Provide an option to commit all state changes to test node once debugging is finished

0.2.4.5 Component: Opcode and Instruction Mappings

A requirement of the emulator is to handle the current position in the bytecode that is being executed.

0.2.5 CodeFile

0.2.5.1 Abstract Specification

0.2.5.2 Interface Design

0.2.5.3 Component: Contract File

0.2.5.4 Component: Contract

0.2.5.5 Component: AST Traversal

0.2.5.6 Component: Source Mapping

0.2.5.7 Component: Compiler

0.2.5.8 Component: Variable Decoding

0.3 User Interface

– What To Do: - Provide a 'Abstract Spec' of every component (Debugger, Compilier, Source Map, AST, Contracts, CodeFile, etc) - provide data structs, and component design of every piece of that (graph it out. Graphs don't have to be as complicated as first. But enough to get the gist of the design of the compnonent) -

Test Rust Code

```
pub trait Foo {
    fn killAllBunnies<F>(fun: F) -> AstItem
    where
        F: Fn(&mut AbstractFunction) -> bool;
    fn scrapeWebForSkynet();
}
```

0.4 Help System

0.5