EDB: Debugger for Ethereum's Programming Languages

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

Andrew Plaza

October 29, 2018

Abstract

1. Hello

Contents

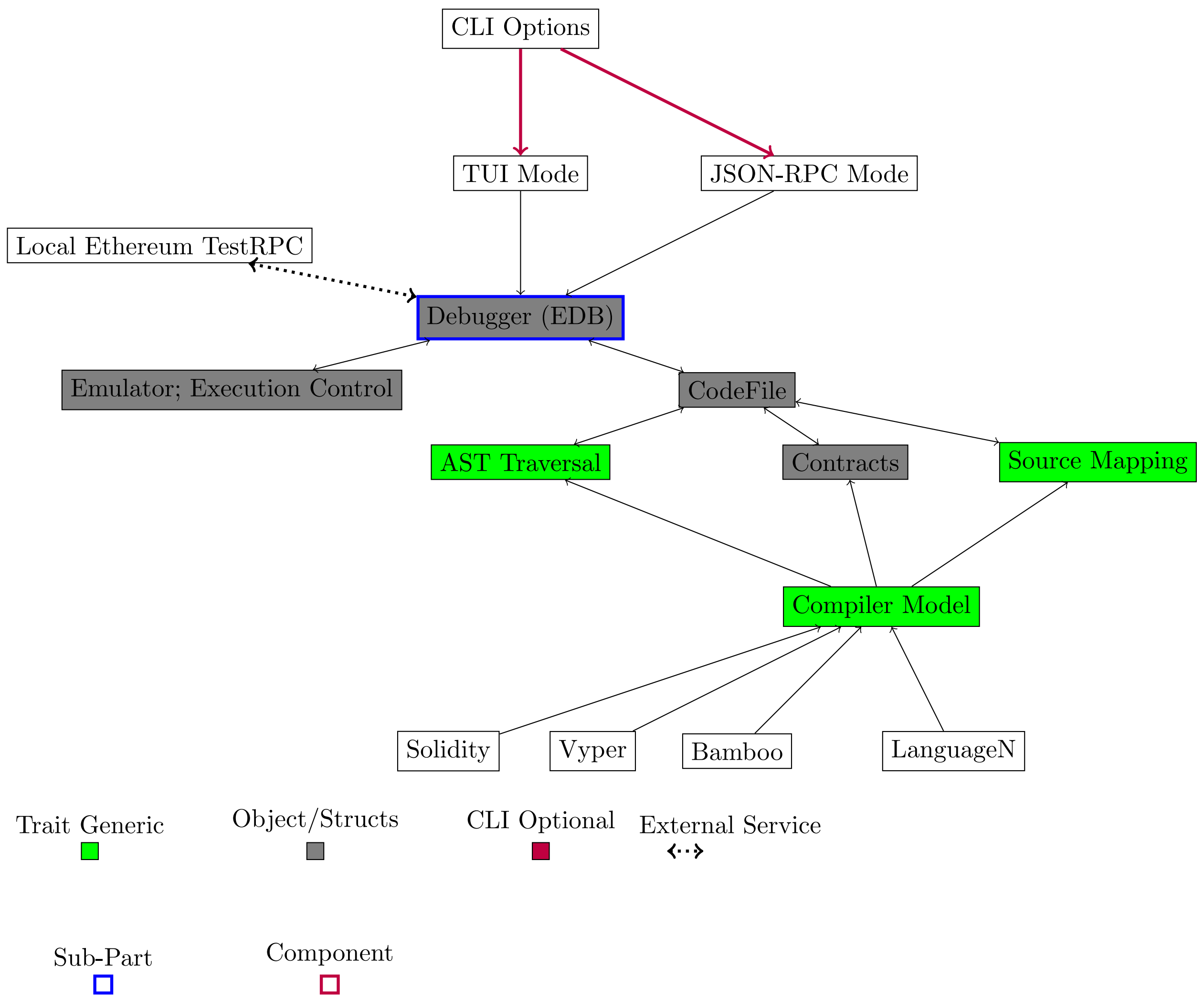
0.1 Introduction

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 Compiler Model and the Emulator model, which make up the functionality of the debugger. The compiler 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 CLI Options: TUI Mode, JSON-RPC Mode

0.2.2.1 Abstract Specification

0.2.2.2 Interface Design

0.2.2.3 Component Design: TUI Mode

0.2.2.4 Component Design: JSON-RPC Mode

0.2.3 Debugger Core

0.2.3.1 Abstract Specification

0.2.3.2 Interface Design

0.2.3.3 Algorithm Design

0.2.3.4 Data Structure Design

0.2.4 Emulator, Execution Control

0.2.4.1 Abstract Specification

0.2.4.2 Interface Design

0.2.4.3 Algorithm Design

0.2.5 CodeFile

0.2.5.1 Abstract Specification

0.2.5.2 Interface Design

0.2.5.3

0.2.6 AST Traversal

0.2.6.1 Abstract Specification

0.2.6.2 Interface Design

0.2.7 Contracts

0.2.7.1 Abstract Specification

0.2.7.2 Interface Design

0.2.8 Source Mapping

0.2.8.1 Abstract Specification

0.2.8.2 Interface Design

0.2.9 Compiler

0.2.9.1 Abstract Specification

0.2.9.2 Interface Design

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