Runtime Verification of Smart Contracts
On the Ethereum network

Lars Stegeman [s1346466]
l.stegeman@student.utwente.nl

August 11, 2018

Contents

1 Abstract

2 Introduction

3 Solidity

4 Tool Overview

5 Annotation Language

The first step towards implementing the tool is to define an annotation syntax, and formally write this down using a grammar.

5.1 ANTLR

5.2 Grammar definition

There already exists a grammar for the complete Solidity language. This grammar is written down using the language that is used by ANTLR tool. ANTLR is a parser generator which helps automate the building of a compiler. ANTLR has the capabilities to extend certain grammars. This is done by inheritance over the original grammar. This principle is explained in detail here (find ref). We will use this principle to extend the grammar of Solidity to recognize the special annotations that will later be used in the tool.
The annotations have certain requirements that can be summarized the following way. Later each requirement is discussed in detail.

- Annotations can be specified at the top level.
- Annotations should be able to reference all variables used in the contract.
- Basic math operations can be used within annotations.
- Annotations can not have side effects.
- The type should be boolean at the highest level (that way they can be verified).
- There are two types of annotations invariants and pre- or postconditions to a function.
- Global variables (`msg` and `address`) can be used in annotations.

The annotation syntax is heavily inspired from the JML annotation syntax. But has a lot less built in keywords since the setting is easier and the tool is less complex. The original grammar is extended in such a way that annotations can only be defined on the top level. The relevant parts of the original Solidity grammar can be seen in the snippet below. This does not include the full grammar specification but only the parts that are relevant for the annotation syntax.

```
grammar Solidity;

sourceUnit
  : (pragmaDirective | importDirective | contractDefinition)* EOF ;

contractDefinition
  : ( 'contract' | 'interface' | 'library' ) identifier
    ( 'is' inheritanceSpecifier (',' inheritanceSpecifier )* )?
    '{' contractPart* '}' ;

contractPart
    : stateVariableDeclaration
    | usingForDeclaration
    | structDefinition
    | constructorDefinition
    | modifierDefinition
    | functionDefinition
    | eventDefinition
    | enumDefinition ;
```

In the original grammar the definition of `contractPart` is what defines the declaration of variables and the definitions for structs and functions. This is where the extra annotations have to be added to the grammar. The snippet below shows the basic definition of an annotation. This is not the complete grammar some of the tokens are omitted from this snippet, since they are not required to understand the grammar definition.

```
grammar SolidityAnnotated;
import Solidity;

contractPart
  : stateVariableDeclaration
  | usingForDeclaration
  | structDefinition
  | constructorDefinition
  | modifierDefinition
  | functionDefinition
  | eventDefinition
  | enumDefinition
  | annotationDefinition ;

annotationDefinition
  : AnnotationStart AnnotationKind annotationExpression ;
```

```
annotationExpression
  : '(' annotationExpression ')'
  | annotationExpression compareOp annotationExpression
  | annotationExpression booleanOp annotationExpression
  | annotationExpression integerOpBoolean annotationExpression
  | annotationExpression integerOpInteger annotationExpression
  | '!'annotationExpression
  | ('\\forall' | '\\exists') '(' identifier elementaryTypeName ':' annotationExpr
  | primaryAnnotationExpression ;

primaryAnnotationExpression
  : primaryExpression
  | primaryAnnotationExpression '.' identifier
  | primaryAnnotationExpression '[' primaryExpression ']'
  | '\\old' '(' primaryAnnotationExpression ')' ;

AnnotationStart
  : '//@' ;

AnnotationKind
  : 'inv'| 'pre'| 'post' ;

LINE_COMMENT
  : '//' ~[@] ~[\r\n]* -> channel(HIDDEN) ;
```

An AnnotationDefinition is composed of multiple components. It has a `AnnotationStart`, `AnnotationKind` and `annotationExpression` component. The `AnnotationStart` token is used to signal that an annotation definition is coming next. This is defined as '//@' making it a line comment to other solidity compilers. This makes annotated solidity code still compilable by normal compilers. For the grammar to accept this notation the `LINE_COMMENT` token has to be adjusted to not accept '@' as a second character. Otherwise all annotation comments would be recognized as a `LINE_COMMENT` making it unusable.
There are two types of annotations they are defined by the token `AnnotationKind`. They can either be a invariant or a pre- or post-condition of a function.
Each annotation has an expression which has to be evaluated called `annotationExpression`. The expression parser rules are separated between `annotationExpression` and `primaryAnnotationExpression`. This is needed to keep the hierarchy in parsing and prevent using primary definitions with complex expressions. For example using the keyword '\old' before parenthesis.
The annotation expressions use a different parser rules than the expression rules that are used within the original Solidity grammar. The `annotationExpression` does not allow syntax like `expression + '++'` and to distinguish these a new parser rule was introduced for annotations only.

6 Validation

6.1 Design

Annotations have to be validated on certain aspects for them to be correct and usable. These aspects have to be verified first for the annotations to be useful in the next generation phase. The parser ensures annotations are syntactically correct however there are more properties that have to be checked. In short the validation step within the program has to do the following things:

- Collect all variables within the contract.
- Typecheck annotations.

The validation phase will consist of two programs that walk the complete parse tree. The first walk will collect all the variables and defined structures and store these in an information object. The second walk will type check each annotation individually. During this type checking the type of each identifier is looked up using the collected information from the first walk.

6.2 Implementation

7 Generation

- TODO