Runtime Verification of Smart Contracts

On the Ethereum network

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

August 8, 2018

Contents

1 Abstract

2 Solidity

3 Tool Overview

4 Annotation Language

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

4.1 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 priciple 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 that is explained later.

```
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.

```
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' ;

booleanOp
  : '&&' | '||' | '->' ;

compareOp
  : '==' | '!=';

integerOpBoolean
  : ('>'|'>='|'<'|'<=') ;

integerOpInteger
  : '+' | '-' ;

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

5 Validation

- Collecting variables
- Typechecking annotations

5.1 Implementation

6 Generation

- TODO