Runtime Verification of Smart Contracts
On the Ethereum network

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

September 14, 2018

Contents

1 Abstract **1**

2 Introduction **1**

3 Solidity **1**

4 Tool Overview **1**

5 Annotation Language **3**
5.1 Grammar definition 3
5.2 Examples 5

6 Type Checker **5**
6.1 Design 5
6.2 Implementation 5

7 Generation **5**

8 Tool Usage **6**

9 Case study **6**

10 Future work **6**

11 Related Work **6**

1 Abstract

2 Introduction

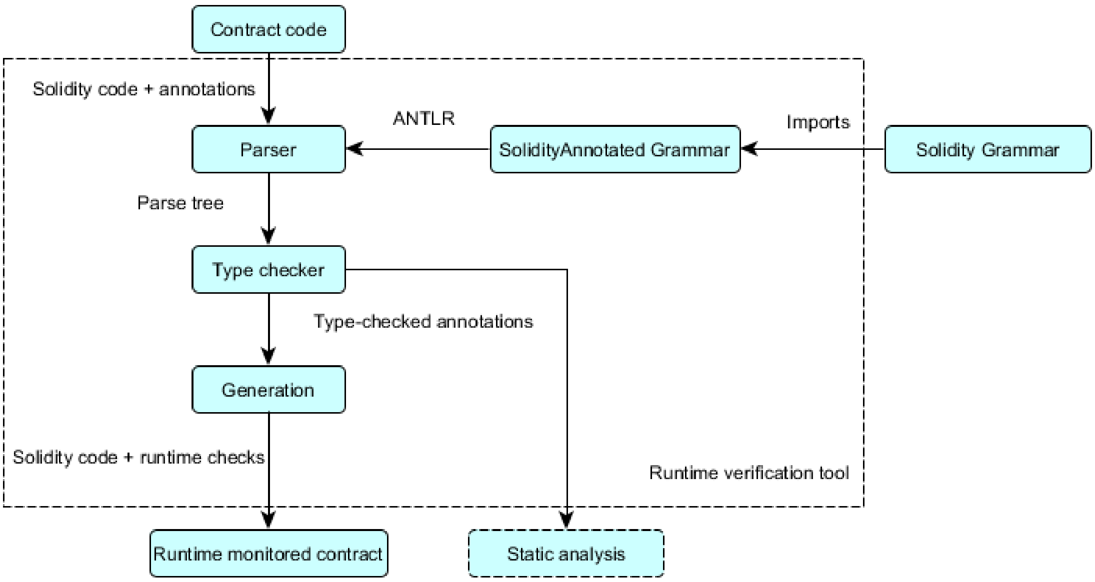
3 Solidity

Section explains basics of solidity in order to understand the rest of the paper. Same section as in ResearchTopics

4 Tool Overview

- Tool works in two phases validation and generation.
- Validation typechecks and checks if annotations are well formed according to grammar.
- Generation generates original solidity code with extra added code to check annotations at runtime.

- Approach uses a Solidity grammar that can be updated easily for future updates.
- Output of validation phase can be used for other verification tools (result is a parse tree).
- Annotations can be defined as invariants and pre or post conditions for functions.
- Annotations use JML like syntax.
- Tool can be used both during development (as extra test cases) or on the actual live blockchain (this probably costs a lot of extra gas = ether).



In the picture the complete overview of the tool can be seen. Within the dashed square the implemented parts are visible. The arrows indicate the flow of the contract code throughout the program.
First contract code has to be annotated according to a specified grammar. Section 5 explains the grammar in more detail and gives some example annotations. The tool ANTLR is used to generate code for the lexer and parser. The result of this step is a parse tree which can be used for later stages.
The next step is type checking the annotations. This uses the parse tree to examine the annotations and check if they are valid. The type checking is done bottom up and works in two phases. The first phase collects all the relevant variables. This includes state variables and function definitions (function name, arguments and return values). The next phase uses this information to do the actual type checking of the annotations. This is explained in more detail in Section 6.
The result of the type checker phase are type-checked annotations. In practice these are parse tree objects in which the types correspond to the operators used and the identifiers that are used are also defined in the contract. This is used as input for the generation phase. The generation phase will operate on the information that is created during the type checker phase. For each annotation it will generate the code that is needed to check it during runtime. This happens in a single passage of the complete parse tree. Details on this phase can be found in Section 7.
Using the annotation grammar all specifications can be expressed. But not all specifications can be translated to code that can be checked at runtime. This means that some annotations will not be translated to runtime monitored contract code. This is the case because of limitations within the Solidity language. For example the mapping type is not iterable and the keys are not known. This means that mappings have to be replaced with other constructions. In the current state of the tool only basic mappings can be replaced with iterable constructions.
The output of the type checker phase can also be used for other static analysis tools. The benefit of using the tool to validate the annotations is that the result is a type checked parse tree that can be parsed and traversed in various ways to be useful for static verification methods.

5 Annotation Language

The first step towards implementing the tool is to define an annotation syntax, and formally write this down using a grammar. The parser generator that is used is called ANTLR. Using the grammar definition the lexer and parser will be automatically generated. The output of this phase is a parse tree that can be used in later stages of the tool.

5.1 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 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 of the contract.
- 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.

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 within 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. `primaryExpression` and `identifier` are parser rules that are defined in the original Solid-

ity grammar. The annotation expressions make use of these rules so that they do not have to be defined again.

5.2 Examples

In this section a couple of annotation examples will be given for example contracts. First a contract snippet is shown and later the meaning of this annotation is explained.

```
uint256 nr1;
uint256 nr2;
//@ inv nr1 >= nr2
```

Defines an invariant that will be checked at the start and end of every function. `nr1` and `nr2` are global contract variables. `nr1` should always be bigger then `nr2`.

```
address owner;
//@ post \old(owner) == owner
function doSomething() public{
    // ...
}
```

Defines a post condition on the function `doSomething()`. Checks if the owner is not changed during execution of the function.

```
uint256[] a;
//@ inv \forall(x in a: a[x] > 0)
```

Defines an invariant that will check if all elements in array `a` are positive.

```
uint256 b;
//@ post (msg.sender == owner) -> (\old(b) != b)
function changeSomething() public{
    // ...
}
```

Postcondition for the funcion `changeSomething()`. If the sender of this transaction is equal to the owner (`msg.sender`), variable `b` must be different from the start of the function.

6 Type Checker

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. The typecheck phase will consist of two phases 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

- Explain generation of functions for annotations. Each annotation will be transformed to a function, the function takes in arguments that are also generated automatically.
- Explain how functions are calling the code for checking annotations. Original Function will be transformed to a private function and renamed to 'functionName"+ _body. New function calls this function with the extra annotations.

- Implementation of generation is using a TokenStreamRewriter (with whitespace) and an abstract parse tree (only tokens that are in grammar). Tokens in both structures know the position, this way the original solidity code can be printed and extra code can be added through traversing the parse tree.

8 Tool Usage

9 Case study

- Take contract that has vulnerability and add annotations
- Show parts of generated code that will throw an event.
- Show difference in gas usage?
- Show how forall annotations are handled (example contract of minimal token).

10 Future work

- The tool cannot handle contract inheritance.
- The tool does not have a mechanism for exposing the parse tree for other programs after validation.

11 Related Work

- ContractLarva: runtime monitoring based on defined states and transitions.
-