UNIVERSITY OF WATERLOO
ECE653 - SOFTWART TEST/QUAL ASSUR/MAINT
PROJECT REPORT

Formal Verification of Solidity Smart Contracts Using Dafny

Report by:
Alireza LOTFI TAKAMI
Student ID: 20854299
Email: alotfita@uwaterloo.ca
Krishna KANTWALA
Student ID: 20868348
Email: kdkantwa@uwaterloo.ca
Aditi PATADE
Student ID: 20868672
Email: apatade@uwaterloo.ca

Instructor:
Dr. Vijay GANESH

August 20, 2020

1 Abstract

A smart contract is a self-executing contract with the terms of the agreement between buyer and seller being directly written into lines of code. The code and the agreements contained therein exist across a distributed, decentralized blockchain network. The code controls the execution, and transactions are trackable and irreversible. Smart contracts permit trusted transactions and agreements to be carried out among disparate, anonymous parties without the need for a central authority, legal system, or external enforcement mechanism [1]. Solidity is a domain-specific programming language intended for smart contract development. It aims to reduce the transaction costs resulting from contract execution on distributed ledgers such as the Ethereum.

Solidity contracts, however, must comply with safety and security standards, which require rigorous inspection and certification [2].This paper demonstrates the implementation of a verification tool to ensure bug-free designing of the smart contract . This tool is expected to take the smart contract written in solidity and translate it to dafny. Dafny being a static program verifier will be used to verify the functional correctness of the contract [3]. We consider correctness as the state of the program without any bugs such that the program observes safe programming practices.

2 Introduction

A blockchain is a decentralised, distributed, and mostly public, digital ledger composed of records called blocks that are used to record transactions through various computers so that any block involved can not be retroactively changed without altering all subsequent blocks. This helps the participants to independently and fairly inexpensively review and inspect transactions. Using a peer-to - peer network and a distributed time stamping server, a blockchain database is managed autonomously They are authenticated by collective self-interest motivated by mass collaboration [4]. The biggest advantage of blockchain is that due to its decentralized nature of the network that operates between all the approved parties, there is no need to pay intermediaries [5].

Smart contracts are executable programs that enable the building of a programmable trust mechanism between multiple entities without the need of a trusted third-party. Smart contracts are often written in a Turing-complete programming language called Solidity, which is not easy to audit for subtle errors [see Abstract][6]. Although the faithful execution of a smart contract is enforced by the consensus protocol of the blockchain, it remains the prerogative of the participating organisations to check the validity of the smart contract , i.e., the syntactic implementation meets the best practises, and validate its fairness, i.e. the code adheres to the agreed higher-level business interaction logic.Although manual auditing of contracts is possible to some degree for accuracy, it still remains laborious and vulnerable to error. In comparison, automated formal auditing requires professional

resources and reasoning. The problem gets worse with the the fact that, unlike other distributed system code, smart contracts are permanent and hard to fix in the case of bugs, regardless of how much money they possess [7].

Therefore, to ensure bug-free smart contract we propose command-line a tool which will verify the smart contract using static verification. Since Smart contracts are coded in an object- oriented programming language called Solidity which does not support static verification, there was a requirement of another programming language in which the smart contracts will be translated and the translated code shall be verified for the potential bugs. This requirement shall be fulfilled by Dafny which supports formal verification.

3 Our Method

As we know ensuring that a smart contract is correct is very important because now days considerable amount of money is controlled by blockchains and smart contracts. Therefore, we can say that smart contracts are a kind of safety-critical systems. As smart contracts failure leads to catastrophic consequences, it is very important to do formal verification on them before deploying them on blockchains. Doing smart contracts verification manually is too time consuming and most of the smart contract developers do not have enough knowledge and expertise in software testing and formal verification. Therefore, creating a tool that do this formal verification automatically could be very beneficial.

In this project we are going to propose a solidity static verification tool. To do this verification on smart contract functions we need to add some queries to the smart contract solidity code and then verify that if the smart contract function satisfies the provided queries or not.

In order to implement the Smart contract verification tool we need to turn smart contract into a state machine or another programming language that have the capability to apply static verification. Hence, we need a tool to parse solidity code and then turn int into our target language or state machine. Here we are going to use ANTLR which will help us to turn the input solidity code into a parse tree in a specific target language like Java or python.

In order to apply the static verification we decided to turn the solidity code into Dafny code. Dafny being a supporter of formal verification tool which will verify the translated code, thus giving an output in the form of a report with a list of detected error, if any. We used Python as target language of ANTLR parse tree to turn the parse tree into Dafny code. In general the input of our Solidity smart contract verification tool is a solidity smart contract function and its output is an equivalent Dafny code which then we can add some queries to it to do static verification on it. In the following section we are going to in detail

explain the tool chain that we have used in this project to implement the solidity smart contract verification tool.

4 Tool Chain

We have used different tools in our project to correctly verify smart contracts. In this section we are going to explain each of the tool which briefly focus on writing contracts in solidity, parse it through the antlr and verify the contracts with some description of the design of Dafny.

4.1 Solidity

Solidity is a programming language which is used for implementing smart contracts. With the help of Solidity, we can write applications that implement self-enforcing business logic such as voting, crowdfunding, blind auctions, and multi-signature wallets. Even we can say that solidity is smart contract specific language[8]. Smart contracts which are implemented in solidity language are self-executing piece of code that are compiled and pushed to the Ethereum blockchain for the execution without requiring centralized or trusted parties. Solidity program is complied to bytecode that is executable on the EVM (Ethereum virtual machine). EVM is basically a runtime environment for smart contracts in Ethereum.

Solidity is a contract specific object-oriented programming language where each contract can have features like objects in OOPS such as common data types, state variables, functions and a contract-specific features including event notifiers, global variables, and modifier clauses of the contract [9]. Consider the simple contract given below:

```
pragma solidity ^0.5.0;

contract MyFirstContract {
   uint balance;

   function set(unit x) public {
      balance = x;
   }

   function get() public view returns (unit){
      return balance;
   }
}
```

As shown in the above code example, any solidity code file ought to begin with a ”version pragma” for the code to use, which is the declaration of the version of the solidity com-

piler. It helps to prevent problems with future compiler versions where code with existing version could behave differently.

Contract can be created with the keyword contract and name of the .sol file. A contract is the basic building unit of Ethereum's decentralized applications where all the methods and variables are the part of the contract. In this example, the contract declares a state variable balance of the unsigned integer of 256 bits and defines the methods set and get that can fetch and update the value of the variable. The solidity compiler then takes this high-level code, breaks it down in simple instructions, and convert it into byte code that is executable on the EVM.

4.2 Antlr

Antlr (Another Tool for Language Recognition) is a powerful parser generator, a tool which helps to create parser for reading, processing, executing or translating structured text or binary files. To parse a typical program using regular expression is not enough specially when program is using recursion but with the help of Antlr it becomes easier, faster and mess free.

Antlr is used to build languages, tools, and frameworks. With the help of Antlr, one can generate a parser that can build and walk parse trees in their specific target language. Antlr helps to define a grammar for the language, data format, diagram or any kind of structure that is represented with text for analyzing [10].

To deeply studying parser we need to initially focus on the lexers also known as tokenizers that take the separate characters and build the logical structure by transforming them into tokens. Antlr uses the grammar to generates code that describes what is legitimate in the language and how the language is structured and convert this grammar file into two parts: parser and lexer. A lexer reads the input stream and turns into tokens and parser associates the tokens with the elements of the grammar.

We have used solidity parser for python which built on the ANTLR 4 grammar file for the solidity programming language which helped to generate the solidity parse tree based on the given contract and grammar file in python and allows us to write code that handles the events from the parser.

4.2.1 Dafny

Dafny is an imperative object-oriented programming language which is used to verify the functional correctness of programs with the help of built-in automatic program verifier. Dafny is using built-in specification construct to support formal verification of program. These specifications include preconditions, post-condition, frame specifications, termination metrics, loop variants and loop invariant which helps to prove correctness of code

[11]. With the help of Dafny, it becomes easier for programmers to write correct code without having any run time errors and also correct in terms of what specifically program supposed to do.

Dafny allows to write verifiable programs that automatically check the correctness of the smart contracts. To prove the correctness of the code Dafny depends on high-level annotations. The result of a chunk of code may be given abstractly which is less complicated and less error vulnerable to write. Dafny generates a proof that the code matches the annotations and lighten the load of writing bug-free code [12]". As the annotations are shorter and more direct it becomes easier to write code. Main concepts of the Dafny are described here.

Method:

For any Dafny programs, one of the basic units is the "method" which is piece of executable code. Consider the following example:

```
method addSub(a:int, b:int) returns (addition:int,subtraction:int)
{
   addition := a + b;
   subtraction := a - b;
   return addition, subtraction;
}
```

This code declares a method to perform addition and subtraction functionality which takes two integer parameter called "a" and "b" as an input and returns results "addition" and "subtraction" in integer. The code for the body of the method contained within the braces can consists of a series of statements including conditions, loops, method calls, return etc. Our method body calculating the addition and subtraction for the given input parameters and returning the result.

Pre and post conditions:

To specify the behavior of the methods, here comes the power of Dafny with annotations. Pre and Postconditions have their own keywords "requires" and "ensures" respectively and can be given as part of the method's declaration. Let's say for the above code example, any non-negative integer addition will always greater than subtraction. To implement this we can add annotations as given below:

```
method addSub(a:int, b:int) returns (addition:int,subtraction:int)
   requires a > 0
   requires b > 0
```

```
   ensures addition > subtraction
{
   ....
}
```

Dafny verifies the code as correct with the help of these conditions because this assumption is enough to ensure that the code in the method body is correct.

Assert:

Assertions declared with the assert statement for testing, debugging or proof purposes which says that a particular expression always holds when control reaches that part of the code. Whenever the Dafny encounters an assertion, it tries to prove that conditions hold for all executions of the code.

```
method testing()
{
   var a:= 20;
   var b:= 10;
   var add,sub := addSub(a,b);
   assert add > sub;
}
```

In the above example, Dafny is able to prove the annotation easily for the method addSub as it is similar to the post-condition given in the method declaration.

Overall, examining the pre and post condition and writing assertions to verify assumptions in Dafny can guarantees that existing code is not broken. To understand more detail version of Dafny can be found at [12].

5 Results

In order to evaluate our approach we have created ten solidity smart contracts. We used these ten smart contract as inputs of our Solidity smart contract formal verification tool. The Solidity formal verification tool then turns the solidity functions into Dafny functions. Hence, The ouptut of our tool is a Dafny function code. Then we can use Dafny code syntax to add queries to to Dafny function and verifiy different variables and parameters of the smart contract function. In the subsequent two sections we are going to explain how this tool have been implemented and we also show some input solidity functions and their equivalent Dafny function with verification queries.

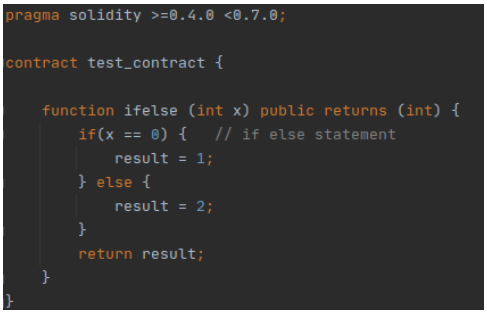
5.1 Experimental Setup

As we explained in last sections, in order to implement the solidity smart contract formal verification tool, we used Antlr, Python, and Dafny. Antler uses an input code and its proper grammar as its inputs and creates a parse tree which can be used for any purpose. In the solidity smart contract verification project we feed Antlr with smart contract solidity code and a proper grammar file. We did not apply any changes on the input grammar file but in future to add more features to the project and make it more flexible we have to apply some changes on the solidity grammar file. So, in our case the output of Antlr is a solidity parse tree. Antlr has the option to create the parse tree for some different darget languages like Python and Java. Figure 1 indicates created parse tree for a simple input solidity function which only contains a return statement.


```
{'children': [{'name': 'solidity',
               'type': 'PragmaDirective',
               'value': '>=0.4.22<0.6.0'},
              {'baseContracts': [],
               'kind': 'contract',
               'name': 'test_contract',
               'subNodes': [{'body': {'statements': [{'number': '3',
                                                      'subdenomination': None,
                                                      'type': 'NumberLiteral'}],
                                      'type': 'Block'},
                             'isConstructor': False,
                             'modifiers': [],
                             'name': 'ifalone',
                             'parameters': {'parameters': [{'isIndexed': False,
                                                            'isStateVar': False,
                                                            'name': 'y',
                                                            'storageLocation': None,
                                                            'type': 'Parameter',
                                                            'typeName': {'name': 'int',
                                                                         'type': 'ElementaryTypeName'}}],
                                            'type': 'ParameterList'},
                             'returnParameters': {'parameters': [{'isIndexed': False,
                                                                  'isStateVar': False,
                                                                  'name': None,
                                                                  'storageLocation': None,
                                                                  'type': 'Parameter',
                                                                  'typeName': {'name': 'int',
                                                                               'type': 'ElementaryTypeName'}}],
                                                  'type': 'ParameterList'},
                             'stateMutability': None,
                             'type': 'FunctionDefinition',
                             'visibility': 'public'}],
               'type': 'ContractDefinition'}],
 'type': 'SourceUnit'}
```


Figure 1: Parse tree of a simple solidity function with only one return statement.

As Python decreases the development time, we used python as Antlre target language. Therefore, in our project the parse tree of Antlre can be used by Python. We used Python to use Antlre's output parse tree and also to create a command-line user interface as interface of our solidity smart contract formal verification tool. As covering all solidity syntax is a time consuming task, we started by solidity basic syntax and turned the solidity parse tree into Dafny code. In next step we added some verification queries to the Dafny code and Then checked to see weather this quires will be satisfied or not. In section 5.2 we



```
pragma solidity >=0.4.0 <0.7.0;

contract test_contract {

    function ifelse (int x) public returns (int) {
        if(x == 0) {   // if else statement
            result = 1;
        } else {
            result = 2;
        }
        return result;
    }
}
```

(a) Solidity code

```
method ifelse(x: int) returns (result : int)
requires x >= 0
{
  if(x == 0){
    result :=1 ;
  }
  else{
    result :=2 ;
  }
  assert result > 0;
  return result;
}
```

(b) Dafny code

Figure 2: Verification of a solidity smart contract function with one if-else statement.

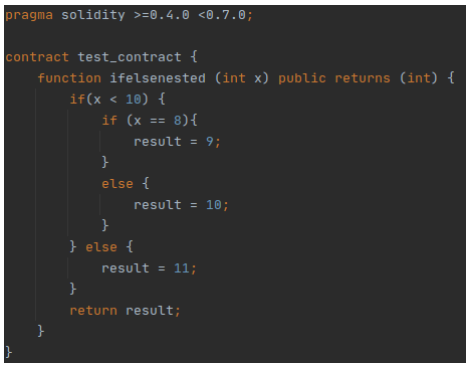
are going to provide the experimental results which are some inputs and outputs of our solidity smart contract formal verification tool.

5.2 Experimental Results

We test the solidity smart contract verification tool using ten different input solidity codes. From the experimental results it can be seen the implemented tool and proposed solidity smart contract verification approach can correctly be used for smart contract verification. As we said we used ten different smart contracts solidity code to check correctness of our approach but here were are going to show two different input solidity smart contracts and their output which were produced by our solidity smart contract verification tool.

Figure 2 shows a solidity smart contract and its equivalent Dafny code which has been produced by our solidity smart contract verification tool. Where Figure 2a show the solidity smart contract function which consist of one if-else statement. We used this solidity code as input to our solidity smart contract verification tool. The equivalent Dafny code of the solidity code is Figure 2b which has been produced by our Solidity verification tool. After producing the Dafny code, we added some queries to it to do the verification. For example we have added a ”require” and an ”assert” statement to the produced Dafny code. We can add any other queries using Dafny syntax to verify any variables in the solidity code.

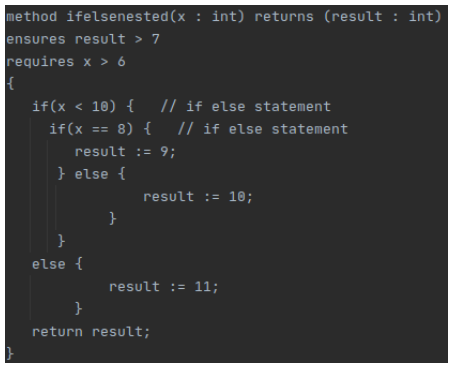
Figure 3 shows another solidity smart contract and its equivalent Dafny code which has been produced by our solidity smart contract verification tool. Where Figure 3a show the solidity smart contract function which consist of a nested if-else statement. We used this solidity code as input to our solidity smart contract verification tool. The equivalent Dafny code of the solidity code is Figure 3b which has been produced by our Solidity verification tool. After producing the Dafny code, we added some queries to it to do the verification. For example we have added an ”ensures” and a ”require” statement to the produced Dafny code. We can add any other queries using Dafny syntax to verify any



```
pragma solidity >=0.4.0 <0.7.0;

contract test_contract {
    function ifelsenested (int x) public returns (int) {
        if(x < 10) {
            if (x == 8){
                result = 9;
            }
            else {
                result = 10;
            }
        } else {
            result = 11;
        }
        return result;
    }
}
```

(a) Solidity code



```
method ifelsenested(x : int) returns (result : int)
ensures result > 7
requires x > 6
{
   if(x < 10) {   // if else statement
     if(x == 8) {   // if else statement
        result := 9;
     } else {
              result := 10;
          }
     }
   else {
            result := 11;
        }
   return result;
}
```

(b) Dafny Code

Figure 3: Verification of a solidity smart contract function with nested if-else statement.

variables in the solidity code. With the same way we have created 8 other solidity smart contract and we verified them using their equivalent Dafny code which were produced by our verification tool.

6 Future Work

As we explained, we started the solidity smart contract verification tool by covering some basic syntax of solidity. In the next step, we need to cover all solidity syntax. Then we can turn any arbitrary smart contract functions into its equivalent Dafny code. Therefore we can verify all smart contracts. Another feature that need to be added to the verification tool is adding query syntax directly to Solidity code. In this way, The smart contract developer does not need to know anything about Dafny and the Dafny queries could be easily added directly to the smart contract code.

References

[1] "Smart contracts definition," https://www.investopedia.com/terms/s/smart-contracts.asp, (Accessed on 05/27/2020).

[2] J. Zhu, K. Hu, M. Filali, J.-P. Bodeveix, and J.-P. Talpin, "Formal verification of solidity contracts in event-b," *arXiv preprint arXiv:2005.01261*, 2020.

[3] "Dafny: A language and program verifier for functional correctness - microsoft research," https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/, (Accessed on 05/27/2020).

[4] "Blockchain - wikipedia," https://en.wikipedia.org/wiki/Blockchain#History, (Accessed on 08/19/2020).

[5] "What are smart contracts? [ultimate beginner's guide to smart contracts]," https://blockgeeks.com/guides/smart-contracts/, (Accessed on 08/19/2020).

[6] J. B. Hardaker, J. W. Richardson, G. Lien, and K. D. Schumann, "Stochastic efficiency analysis with risk aversion bounds: a simplified approach," *The Australian Journal of Agricultural and Resource Economics*, vol. 48, no. 2, pp. 253–270, 6 2004.

[7] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." in *NDSS*, 2018, pp. 1–12.

[8] "Solidity — solidity 0.6.8 documentation," https://solidity.readthedocs.io/en/v0.6.8/, (Accessed on 08/19/2020).

[9] "Introduction to smart contracts — solidity 0.6.8 documentation," https://solidity.readthedocs.io/en/v0.6.8/introduction-to-smart-contracts.html, (Accessed on 08/19/2020).

[10] "Antlr," https://www.antlr.org/, (Accessed on 08/19/2020).

[11] "Dafny: A language and program verifier for functional correctness - microsoft research," https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/, (Accessed on 08/19/2020).

[12] "rise4fun," https://rise4fun.com/Dafny/tutorial/Guide, (Accessed on 08/19/2020).