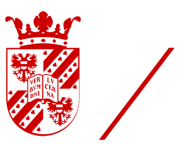
M^3 Source Code Model for Ethereum Solidity

BACHELOR THESIS

Student: Tobian Koenen (S4093933)
Supervisor: Tijs van der Storm



university of groningen

July 2024

Abstract

Modern software systems are becoming more complex than ever and this means that there is a dire need for re-engineering tools that make these systems more manageable. This thesis contains the development of a Solidity specific source code model using the M^3 feature of the Rascal programming language. Solidity is a programming language that is used in the development of smart contracts, which usually involves a lot of money and thus requires precision, making tools that improve code written in it quite valuable.

Acknowledgement

I would like to thank my supervisor, Tijs van der Storm, for helping me get on the right track whenever I had questions.

List of Figures

Listings

List of Tables

Contents

1 Introduction

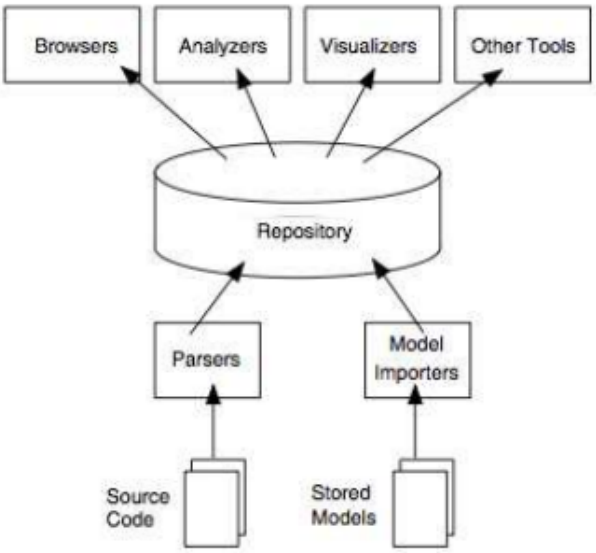
1.1 Problem

Modern society requires software systems to be bigger and more complicated than ever, making them harder to maintain and extend upon by the day. When they do not evolve with time, they are eventually rendered useless. The re-engineering of these systems, however, is quite impossible without the appropriate tool support. Tools to extract facts from languages, analyze, display metrics and compute reports can be used to overcome the daunting task of revamping sizeable software systems. The structure of a re-engineering environment can be seen here (see figure 1) [10]. It generally consists of an abstracted model of the source code, called a source code model. This model describes what and in which way information is modeled, making it easier to perform the aforementioned tasks.

As programming languages may differ a lot from each other, these source code models need to be fine-tuned for each language to ensure accurate modeling. Solidity, a relatively newer programming language used for the development of smart contracts, is a language that particularly values clean source code. This is because the transactions made with these smart contracts run up to millions of dollars in funds, making bugs undesirable as it could lead to big losses. As these smart contracts need to be reliable, it might be of interest to analyze programs coded in Solidity, but a proper bridge between the language and analysis tool was still lacking.

Hence, the development of a source code model for the Solidity programming language. This is done through M^3[8], a general model for code analytics in Rascal [12].

Figure 1: General structure of re-engineering environment.



1.2 Research questions

The research questions in this thesis are structured in the form of a main question along with sub-questions:

- How do we obtain a M^3 model for Solidity?
  - How do we obtain the necessary data to populate the model?
  - How do we convert this data to Rascal data types?
  - How do we populate the model?

1.3 Objectives

While keeping the before mentioned research questions in mind, the objectives of this thesis are firstly to obtain the data required to populate the model. This, however, has to be achieved in the most efficient manner attainable, implying that all the information about a Solidity program cannot be manually extracted from it as typing everything in will consume too much time.

The extracted information will have to be converted to Rascal data types in order to fit the model. Once the model has sufficient information, analysis on Solidity programs can be performed to prove it works.

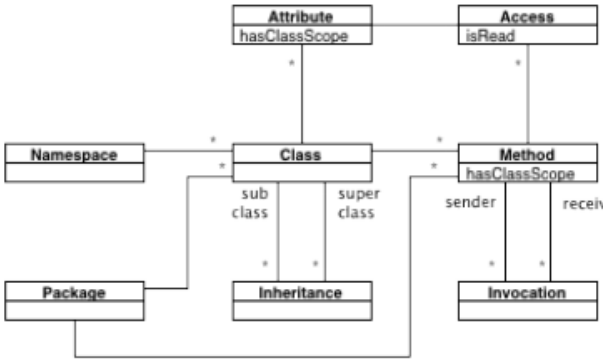
For example, the cyclomatic complexity of Solidity functions can be calculated. This is done through counting the number of linearly independent paths in a function. Functions with a high cyclomatic complexity are more prone to bugs, and so it might be favorable to refactor or redesign them entirely.

It will also be interesting to know whether it is possible to detect cyclic dependencies in the Solidity programs with the M^3 model. A cyclic dependency is a code smell where two modules or components in a program depend on each other. This makes trying to understand the code a lot harder also increasing the difficulty of maintaining and refactoring. The detection of these cycles can simplify the search for cyclic dependencies, which can then be eliminated from the code.

2 State of the art

Various tools that use source code models for the purpose of analysis are available. One of such tools is FAMIX [10], which is a collection containing models that can be viable for the representation of components that are embedded in software systems. With the main purpose of analysis, it provides a rich API featuring the navigation and querying of source code. It is important to note that the goal of FAMIX is to portray programs in a language independent manner. While it is feasible to accomplish this for languages with common subsets, there are certain caveats, including the possibility of easier solutions for some languages, but more importantly: the main source code model (see figure 2) requires extension to support language specific features. Models that have been implemented thus far are: SourceAnchor, Java, File and C. Which means that it does not support solidity yet.

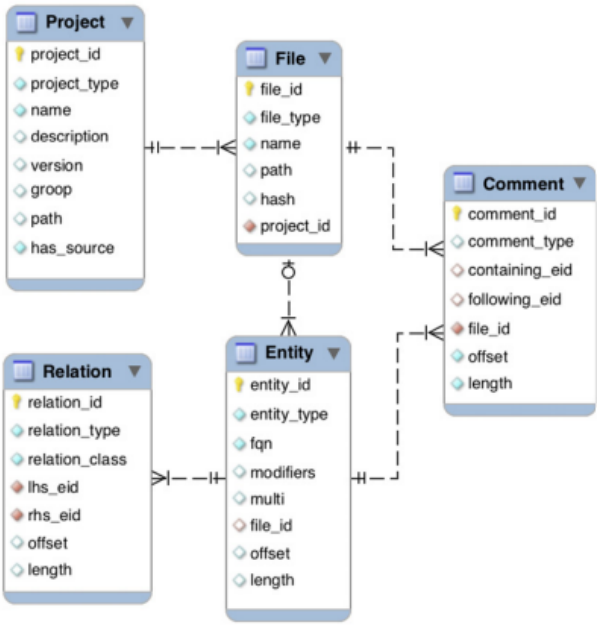
Figure 2: FAMIX-core, the main source code model which all other models are build upon.



Another alike tool is Sourcerer [7] which differentiates itself from FAMIX by placing the emphasis on the collection of source code instead of analysis. The source code collection component of Sourcerer, however, is not of interest as the focus lies on source code models. Because of this emphasis on source code collection, a Java specific model employing an entity/relationship style (see figure 3) is used as opposed to the more generic model of FAMIX, since this enables a more concise representation of specific features of Java and simplifies the mapping to a relational database, allowing for easier source code collection. Hence, this tool is limited by only supporting a single language, highlighting the potential challenge of designing source code models that are applicable across a diverse array of languages.

The challenges encountered in the design of source code models that accommodate a broad spectrum of languages for the use of source code analysis are undeniably complicated. A clarification on the hurdles in this realm can be acquired by investigating the development of the tool that is utilized in this thesis,

Figure 3: The relational model used in Sourcerer.



being M^3 [8]. M^3 is composed of an abstract syntax tree (AST) layer that resorts to a standard interface which is anticipated to be language specific, in order to retain as much accuracy as possible, and a more abstract relational layer made for reuse (see figure 4). The latter supporting variability of languages, as well as integration of metrics across languages for further analysis. Do keep in mind that the design prioritizes accuracy over reusability, as an entire language independent model would produce incorrect metrics. The use of a language specific AST enhances flexibility, and it shows, as out of the source code model tools that have been mentioned up to this point, M^3 is the one that currently provides support for the highest quantity of languages, allowing for source code analysis in Java, PHP, JavaScript, C/C++, Python and Ada[13][2], nonetheless, a lack of support for Solidity.

Regarding Solidity analysis, multiple frameworks that focus solely on this matter exist, such as Smartbugs[9], Slither[11], SIF[14] and many more [16][17]. One of which, namely SIF, also uses an AST in its implementation, making it similar to M^3 in that particular aspect (compare figure 4 & 5). As these frameworks are specifically tailored for Solidity, they might be more accurate than an implementation in the source code models that try to accommodate numerous languages.

A model made entirely for Solidity analysis does exist and makes use of symvalic value-flow static analysis as opposed to just static analysis that all other examples make use of. It is made by the University of Athens and the details can be found here [15]. This particular style of source code analysis results in remarkable accuracy yielding high-value vulnerability warnings with which a significant real-world impact has been made as already several millions of vulnerable funds have been rescued.

Figure 4: An overview of M^3, which produces metrics from source code.

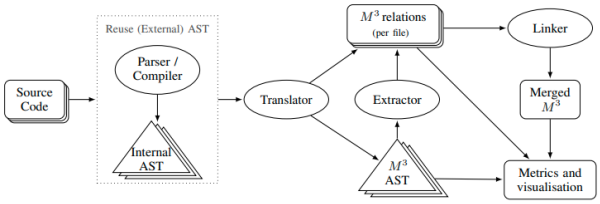
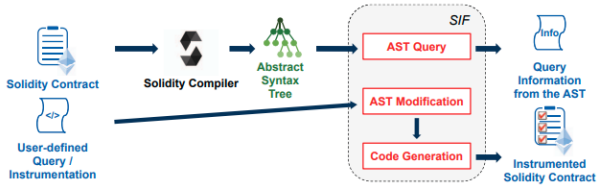


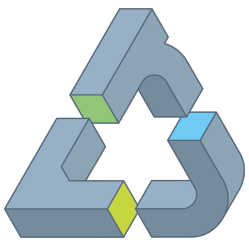
Figure 5: The SIF work flow.

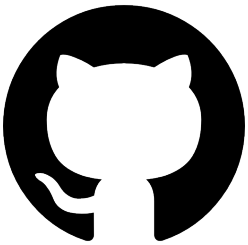


3 Methodology

The majority of the code is written in the Rascal programming language, using its M^3 feature to create a source code model that can analyze programs made in the Solidity programming language. As mentioned in 1.2, the facts about the source code cannot be manually extracted from the programs, as this is an unnecessary amount of work and does not make sense since computers are used to automate processes. Conveniently enough, the standard Solidity compiler solc has an option that can dump the abstract syntax tree of the program it compiles. This can then be converted to Rascal data types and the AST layer [4] of the source code model is completed, which means that the cyclomatic complexity can be calculated. The relational layer, called the M^3 core [5], consists of several fields that can be populated. Two of them being the containment and uses of a program, which is useful to detect cyclic dependencies between components in the code and between modules. This information can be extracted from the AST, as the relations between the nodes and their children are equal to that in the containment. When using these two analysis methods, neither the AST layer nor the relational layer will have to be extended to fit the Solidity programming language, as both the layers have sufficient encapsulation. For other analysis methods that make use of inheritance, for example, the relational layer will need extensions similar to the Java M^3 model [6].

The project will be safely stored on GitHub, which provides a version control mechanism to prevent unintended development errors, as well as a cloud-based storage to mitigate the risk of data loss in case of hardware failures. The code itself will be written in Visual Studio Code (VSCode) as this provides an extension that is specifically made for Rascal, creating an easy-to-use environment for the development of Rascal projects.



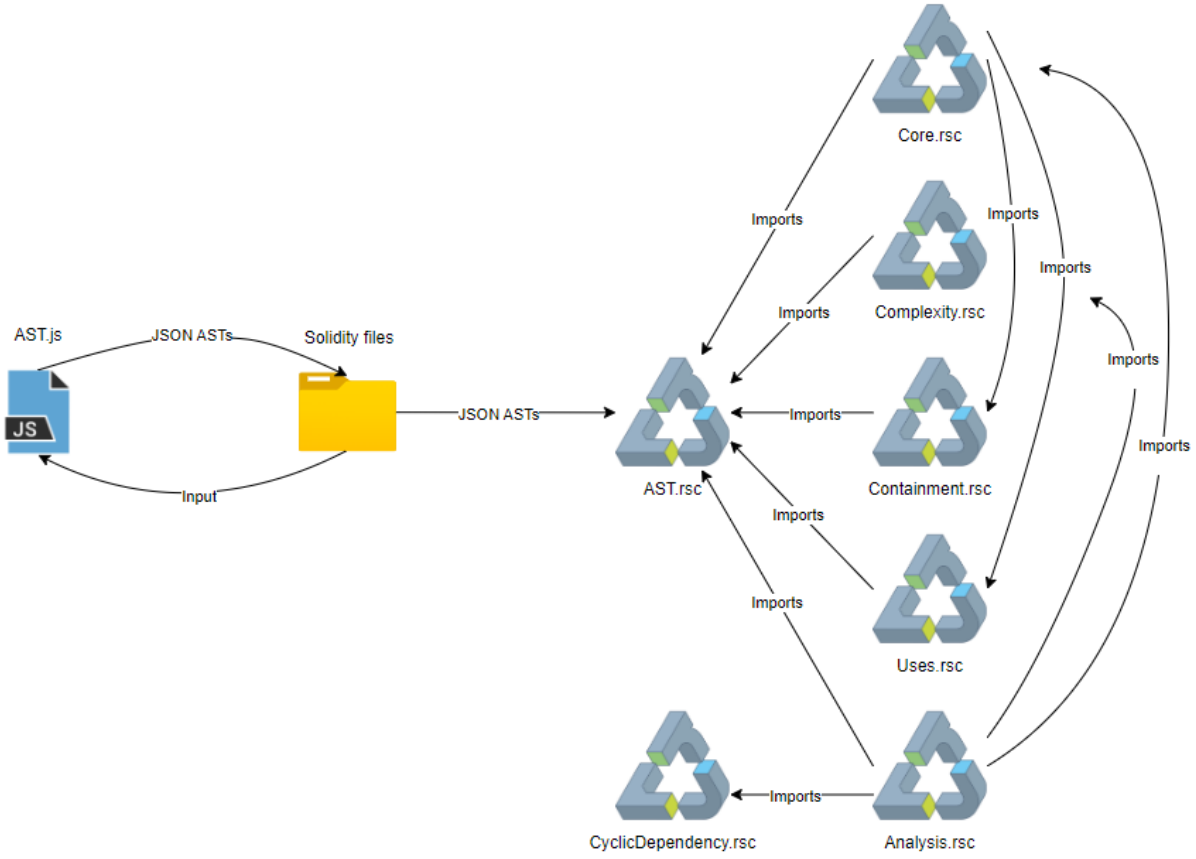


4 Implementation

4.1 Architecture overview

In Figure 6 an overview of the architecture can be found. The first step is to find a directory to analyze. This directory will serve as the input for a JavaScript that generates JSON ASTs of the Solidity files in the directory. These JSON ASTs can then be found by the AST module in Rascal, which converts these into Rascal ASTs. Other modules in the library import the AST module to access and manipulate the data to further populate the model. More details on these steps are found in the sections hereafter.

Figure 6: Overview of the architecture.



4.2 Generating JSON ASTs

A directory is given as input to a JavaScript, which performs a few steps to generate a JSON AST for every Solidity file in the directory. It starts by recursively traveling through the folder structure of the directory to find all files that use a .sol extension, the paths of these files are saved to an array. Each file in the array is then compiled with the `--ast` flag, the output of the compilation is saved to a new file with the same name and location as the Solidity file it represents but with an AST.json extension. Here is part of a JSON AST to get a better picture of what is generated:

```
{
  "absolutePath": "Proxy.sol",
  "exportedSymbols": {
    "Proxy": [
      43
    ]
  },
  "id": 44,
  "license": "AGPL-3.0",
  "nodeType": "SourceUnit",
  "nodes": [
    {
      "id": 1,
      "literals": [
        "solidity",
        "^",
        "0.8",
        ".0"
      ],
      "nodeType": "PragmaDirective",
      "src": "38:23:0"
    },
    {
      "abstract": true,
      "baseContracts": [],
      "canonicalName": "Proxy",
      "contractDependencies": [],
      "contractKind": "contract",
      "documentation": {
        "id": 2,
        "nodeType": "StructuredDocumentation",
        "src": "65:296:0",
        "text": " @title Proxy\n @dev Implements delegation of calls to
            other contracts, with proper\n forwarding of return values
           and bubbling of failures.\n It defines a fallback function
           that delegates all calls to the address\n returned by the
           abstract _implementation() internal function."
      },
      "fullyImplemented": false,
      "id": 43,
      "linearizedBaseContracts": [
        43
      ],

      ... and another 720 lines
```

Listing 1: Code snippet of ProxyAST.json.

4.3 Populating the AST layer

When the JSON ASTs are present within the target directory, the Rascal module AST can find them and can start converting them to the Rascal data types. The JSON ASTs are first converted to strings, then to maps, from which the individual JSON nodes that contain the code components of the Solidity program can be extracted. The Solidity grammar [3] has been entirely defined in Rascal data types and a part of it can be seen below, it consists of the data types: Declarations, Expressions, Statements and Types. All the nodes are parsed, where each node is mapped to a Rascal data type based on the type of the node, creating the AST layer of the model.

```
// Solidity grammar defined in rascal data structures
data Declaration
    = \pragma(list[str] literals)
    | \import(str path)
    | \contract(str name, list[Declaration] contractBody)
    ...
    ;

data Expression
    = \binaryOperation(Expression left, str operator, Expression right)
    | \unaryOperation(Expression expression, str operator)
    | \assignment(Expression lhs, str operator, Expression rhs)
    ...
    ;

data Statement
    = \block(list[Statement] statements)
    | \uncheckedBlock(list[Statement] statements)
    | \variableStatement(list[Declaration] declaration, Expression expression)
    ...
    ;

data Type
    = \int()
    | \uint()
    | \string()
    ...
    ;
```

Listing 2: Code snippet of AST.rsc.

4.4 Populating the relational layer

The AST layer now contains all information about the Solidity programs, and so this module will need to be imported in order to access this information. To populate the containment, the Rascal ASTs will have to be traversed. Each node in the Rascal AST has the source location of the component it represents in the Solidity file. As it is already in tree form, every parent node source location with a child node source location represent a containment relation and are added to the model. To populate uses, the tree can also be traversed to find all the import declarations, these declarations contain a string that represents the relative path to the module it is importing. But this is not the location of the import file yet and so this relative path is first converted to the absolute path. The relation between the file that imports and is being imported can now be added to uses.

4.5 How to use

To use this tool for the analysis of Solidity programs, both Solidity and Rascal code need to be able to compile properly. The configuration used in this thesis requires the installation of Node.JS with which the Solidity compiler can be installed as such: `npm install solc`. To run the Rascal code the Rascal extension in VSCode can be installed. Once the installation is complete, the first step is to find a directory to analyze and store this in a folder called 'Github', otherwise the folder and file locations in

the containment part of the relational layer will not be correct. The next step is to generate the JSON ASTs which can be accomplished by opening a terminal in the same folder as the JavaScript AST.js and typing `node AST.js <path>` where path is the path of the directory. The script will output the path of the directoy in Rascal format, which can then be used in Analysis.rsc for the following three commands: `complexity(<path>);`, `cyclicDependency(<path>);`, and to run them both at the same time: `analyze(<path>);`.

5 Evaluation

5.1 Obtaining the M^3 model

The acquisition of the M^3 model can already be deduced from the sections prior to this, nevertheless a concise answer will be given here. The Solidity compiler being able to dump the AST of Soldity programs is a very useful feature that provides all the information needed to populate the whole model. The JSON AST can be mapped one to one to the AST layer of the model by converting it from JSON to Rascal data types and the AST can be traversed to populate the relational layer as is done with containment and uses. All the other fields in the relational layer besides containment and uses, which are implicit declarations, types, messages, names documentation and modifiers can also be derived from the AST for potential further analysis.

5.2 Evaluating the model

Evaluating the model is done by utilizing a variety of metrics, for each metric an explanation is given.

- Completeness, does the model capture all elements of the Solidity language?

  During the mapping of the JSON AST to the Rascal data types, important information relevant to the analysis that was performed such as names, arguments and parameters are extracted. This means that there is still leftover information in the JSON AST and to reach full completeness of the AST layer all information would have to be extracted from the JSON AST. Only containment and uses are populated in the relational layer and moreover, features of Solidity such as inheritance and method overrides would need extension of the relational layer. So neither this layer has achieved full completeness.

- Accuracy, does the model accurately represent the information extracted from the JSON AST?

  None of the information that is taken from the JSON AST is malformed or lost during the process of converting to Rascal data types. Implying that the AST layer represents the information of the Solidity programs accurately. The population of the relational layer does involve data manipulation of the extracted information. All data types within the relational layer are locations however, and during the analysis of cyclic dependencies there were no non-existent locations found, so the relational layer is also accurate.

- Performance, can the model handle large or complex Solidity programs efficiently?

  The performance has been tested by measuring the time it takes to run the script that generates the JSON ASTs and the time it takes to perform both analyses. The samples are the same 20 repositories that are used for the analysis results later on. The results of the performance test can be seen below (see table 1). The most noteworthy thing is the time it took to run the script for repository 1, which took 19 minutes and 58 seconds to finish. The reason for this is that some of the Solidity files in repository 1 are so huge that the compiler runs out of memory. The size of the files is the main factor that influences the performance of the script, as repository 10 with

643 files only takes 3.3s to finish while repository 20 with only 180 files takes 38.1 seconds and this makes sense. The bigger the file, the more code the compiler needs to parse. When not including the outlier that is repository 1, both the script and the analyses finish within one minute, which is a respectable time.

Repository	**.sol files**	**Script**	**Analyses**
1	285	19m58s	4.6s
2	132	10s	1.3s
3	116	3.1s	1.3s
4	100	0.9s	0.6s
5	169	8.7s	2.0s
6	246	21.6s	1.6s
7	244	19.8s	0.8s
8	18	0.8s	0.3s
9	833	57.3s	15.2s
10	643	3.3s	2.3s
11	908	6.6s	2.6s
12	175	2.8s	0.6s
13	60	1.9s	1.3s
14	72	2.7s	0.9s
15	430	8.5s	3.4s
16	75	1.8s	0.5s
17	148	5.8s	0.6s
18	104	5.3s	0.8s
19	185	6.3s	1.1s
20	180	38.1s	2.9s

Table 1: Results of performance measurements.

- Usability, is the model easy to understand and use?

  Using the model is relatively easy, as the installation of the necessary software is minimal. Once a Solidity repository has been imported, there are five commands that can be used to complete the analysis as can be seen in section 4.5.

- Extensibility, can the model easily be extended?

  This is more a feature that belongs to the Rascal programming language. Both the AST layer and the relational layer try to fit most of the commonalities between programming languages, but provide an option for extension when there are language specific features.

- Complexity, how complex is the model?

  The project contains a total of 8 files, each file accomplishing a different task to further complete the model. The lines of code come in at approximately 800 lines, as can be seen in the table below (see table 2). Within these 800 lines of code a total of 29 functions can be found. So it is neither the biggest nor the smallest of projects. The facets that increase the complexity most are the prominent use of switch statements and sequential use of built-in functions in some parts of the code, requiring a bit of thinking to fully understand.

Language	Files	Blank	Comment	Code
Rascal	7	65	49	712
JavaScript	1	17	9	75

Table 2: Counted lines of code.

5.3 Cyclomatic complexity

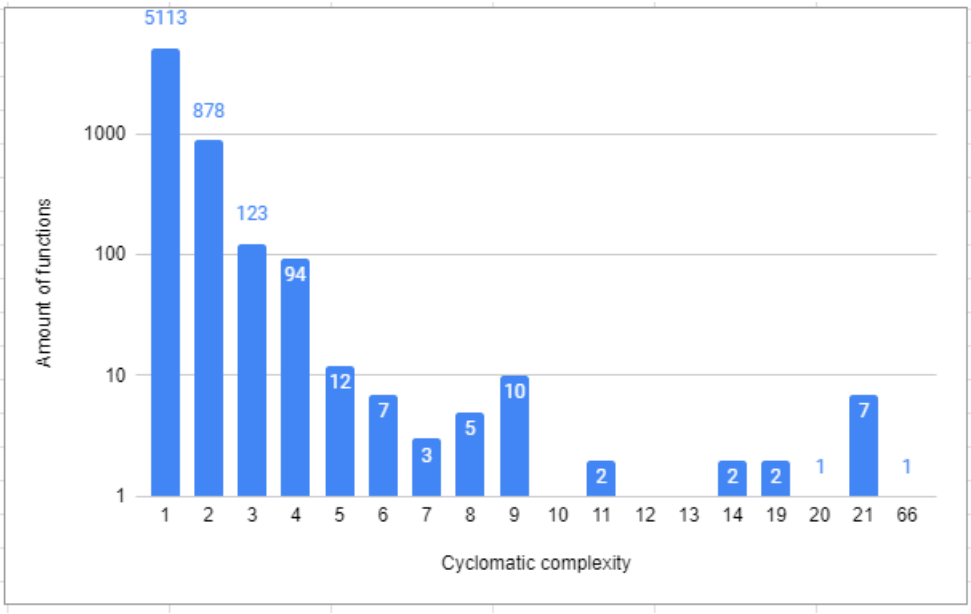
For the calculation of the Cyclomatic complexity the traversal of the Rascal AST is needed once again. This time to find the amount of loops and if statements within a function. For every function found in the AST, it will traverse the statements within and count the loops and if statements, as these increase the amount of linearly independent paths in a function. The cyclomatic complexity of a program starts with a singular path, and so the count always starts at one.

The analysis is done on a total of 20 repositories, these can be found in section 7.2. Most of the repositories belong to that of so-called Decentralized Finance (DeFi) protocols, decentralized applications that innovate upon traditional financial services using blockchain technology. This is because these protocols provide a lot of different financial services that all require smart contracts and the more smart contracts in a repository, the more functions there will be, making cyclomatic complexity analysis more meaningful. The output of the analysis for repository 17 looks like this:

```
Amount of functions:181
Complexities of functions:[165,11,4,0,0,1]
```

The repository houses a total of 181 functions of which 165 have complexity one, 11 functions with complexity two etc. Now when the total of 20 repositories are combined, a whopping 6260 functions are found. The repositories have between 30 and 1444 functions with on average 313 functions. The results of the selected 20 have been combined and put into a bar chart (see figure 7). The chart projects along the y-axis or the height of the bars the amount of functions and on the x-axis the cyclomatic complexity. The most noticeable characteristic of the chart is that the bars are very skewed towards the left. Functions that have a complexity of 1 are the most common by far, with 5113 out of 6260 or around 82% of the functions. This big difference causes the bars that count the complexities from 5 until 66 to disappear if the chart is not put on logarithmic scale and is even still the case for complexities 20 and 66 as these are the complete outliers with only one function to each.

Figure 7: Cyclomatic complexity analysis of 20 repositories (logarithmic scale).



There are multiple explanations as to why Solidity functions are not of high complexity, the most important reason being security. As Solidity programs run smart contracts that can have very high monetary value, Solidity developers often adhere to practices that enhance the safety, meaning they prefer simple functions that only perform one task. This in turn reduces the cyclomatic complexity of the functions they write and decrease the chance of vulnerabilities appearing, as this makes it easier to understand. A second reason for the low complexity being gas fees, functions with high cyclomatic complexity can be more expensive in terms of its gas fees, which are similar to transaction fees, and so functions with small complexity are preferred to safe money on fees.

The most amount of functions are distributed among complexities 1 up to and including 4, meaning that anything with a complexity of 5 and onwards might be worth looking into for a potential re-engineering job. For this reason the complexities are stored in a list of tuples, as a simple print statement can get the locations of these functions:

```
iprintln([location,complexity | <location,complexity> <- complexities, complexity>4]);
```

When analyzing a repository now, for example repository 9, the output will be:

```
[|file:///.../contracts/Misc_AMOs/kyberswap/KyberTickMath.sol|(1347,2799),
21,
|file:///.../contracts/Misc_AMOs/kyberswap_v2/TickMath.sol|(1342,2799),
21,
|file:///.../contracts/Uniswap_V3/libraries/TickMath.sol|(1377,2618),
21,
|file:///.../contracts/Uniswap_V3/libraries/Testing.sol|(1231,2616),
21,
|file:///.../contracts/Misc_AMOs/kyberswap_v2/libraries/TickMath.sol|(1342,2799),
21,
|file:///.../contracts/FPI/ABDKMath64x64.sol|(16057,6983),
66]
```

The files that have a complexity of higher than 5 can now be clicked on and edited. When clicking on the file with complexity 66 one encounters this:

```
function exp_2 (int128 x) internal pure returns (int128) {
    unchecked {
      require (x < 0x400000000000000000); // Overflow

      if (x < -0x400000000000000000) return 0; // Underflow

      uint256 result = 0x80000000000000000000000000000000;

      if (x & 0x8000000000000000 > 0)
        result = result * 0x16A09E667F3BCC908B2FB1366EA957D3E >> 128;
      if (x & 0x4000000000000000 > 0)
        result = result * 0x1306FE0A31B7152DE8D5A46305C85EDEC >> 128;
      if (x & 0x2000000000000000 > 0)
        result = result * 0x1172B83C7D517ADCDF7C8C50EB14A791F >> 128;


      // ... and 62 more if statements


      result >>= uint256 (int256 (63 - (x >> 64)));
      require (result <= uint256 (int256 (MAX_64x64)));

      return int128 (int256 (result));
    }
  }
```

Listing 3: ABDKMath64x64.sol

A function having 65 if statements (remember that cyclomatic complexity is decision points plus one) is not ideal and it should definitely be considered for re-engineering.

5.4 Cyclic dependency

To detect cyclic dependencies between code components and modules, the same method is used because both containment and uses are in the form of a set that hold relations. A directed graph is created by extracting all the individual nodes from the relation set (either containment or uses) and using this relation set as the edges. With the nodes and edges, depth first search can be used to construct all possible paths which might lead to potential cycles.

The same 20 repositories have been used for this. The output of the function for repository 6 looks like this:

```
Cyclic Dependency:
{}
Import cycles:
{
    {
    |file:///.../contracts/libraries/svgs/Profile/Body/BodyShibuya.sol|,
    |file:///.../contracts/libraries/svgs/Profile/Hands.sol|,
    |file:///.../contracts/libraries/svgs/Profile/Body.sol|
    },
    {
    |file:///.../core/contracts/libraries/svgs/Profile/Hands.sol|,
    |file:///.../contracts/libraries/svgs/Profile/Body.sol|
    }
    ...
}
```

Cycles in the code components are shown by the first set that is shown, which is empty in this case, and the import cycles can be seen right below there. This repository has two import cycles where one cycle is a $A \to B \to C \to A$ between three files and the other is a $A \to B \to A$ cycle between two files.

The results of all 20 repositories were somewhat unexpected, but after some research, quite explanatory. From all the 20 repositories, not a single cyclic dependency was found within the code components of the Solidity programs. The first thing that comes to mind is a bug in the function that creates the containment relations, but after thoroughly testing it that was not the case, because when adding a cycle to the containment manually it does print the cycle:

```
containment:
{<|file:///.../contracts/showcase/Game.sol|(3036,23),
|file:///.../contracts/showcase/Game.sol|(3036,10)>};

containment +=
{<|file:///.../contracts/showcase/Game.sol|(3036,10),
|file:///.../contracts/showcase/Game.sol|(3036,23)>};

Cyclic dependency:
{{
|file:///.../contracts/showcase/Game.sol|(3036,23),
|file:///.../contracts/showcase/Game.sol|(3036,10)
}}
```

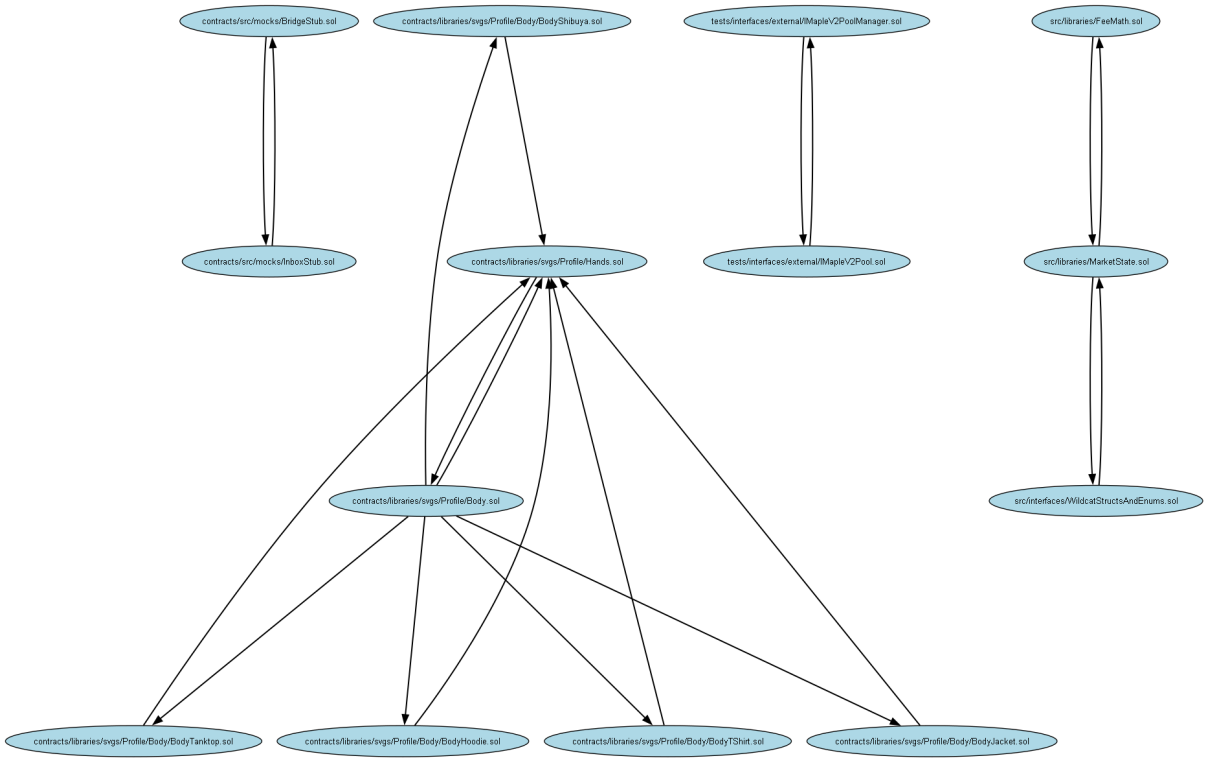
As this is not the problem, further investigation was necessary. After reading through the Solidity documentation [1], the following was found:

> If a contract wants to create another contract, the source code (and the binary) of the created contract has to be known to the creator. This means that cyclic creation dependencies are impossible.

So cyclic dependencies in the creation of contracts is impossible. The lack of cyclic dependencies in functions can be explained by the previous section. Functions in Solidity are usually not complex and so calling one function from another is not common as the safety factor is just too important for a language like Solidity.

Import cycles do definitely exist, as the analysis of the 20 repositories lead to the following results. For both the import cycles between three and two files, a total of 5 occurrences have been found as can be seen in figure 8. The fact that import cycles are more common can also be explained by the concern for safety. Import cycles are not as severe as having cyclic dependencies in functions as these are part of the execution and can introduce some severe vulnerabilities, while import cycles only make the code more complex to understand, allowing for potential vulnerabilities elsewhere. Again, the files that are part of the cycle can be clicked on, allowing for easy access and a quick inspection to consider a possible re-engineering job.

Figure 8: All import cycles that have been found.



6 Conclusion

6.1 Summary

Modern systems require modern solutions. Re-engineering tools for complex and big software systems are becoming necessary to maintain and extend existing software hence this thesis shows an efficient method of obtaining a Solidity M^3 model in Rascal that can be used to analyze Solidity projects. By using a data dump from the Solidity compiler, the information about the source code is easily acquired and can then be converted to Rascal data types to populate the model. The model is not populated entirely, but rather the parts that were required for the analysis that followed, being the analysis of functions and their cyclomatic complexity along with the detection of cyclic dependencies.

6.2 Contributions

The work that is presented in this thesis contributes to the area of study, being source code analysis, in the following ways:

- A detailed methodology on the acquisition of a M^3 model for Solidity is provided and can be used as a reference for potential M^3 models in other languages.
- The tool that has been created for this research can be used by Solidity developers to improve their code by simplifying functions with high complexity and eliminating import cycles.

6.3 Limitations

A limitation that has been encountered during the development of the tool is that the generation of the JSON ASTs does not work for all Solidity files. The compiler installed by Node.js is one for the newer versions of Solidity, and so the programs need to be of the same version or higher. This limits the amount of repositories that can be analysed to only recently updated ones. It could be solved by installing an older version of the compiler, but since old repositories are not really worth it to analyse as they are most likely not being used anymore, the initial compiler version is used only. It also does not work for files that are big enough to make the compiler run out of memory, which could be fixed by giving the compiler more memory to work with. The biggest limitation of the model itself is that it is not entirely complete, as completing it entirely was out of the scope of this thesis. This limits the amount of analysis methods that can be performed with the model, however, with a bit of creativity the model could be used for more than just the analysis of cyclomatic complexity and cyclic dependencies.

6.4 Future work

Currently, the model is not completely populated, nor are concepts such as inheritance and method overrides included. In the future, the model could be further improved by filling in the missing data in the relational layer as well as extending it in a way similar to the Java M^3 model. This way Solidity specific features are also included, which enhances the precision of the model as it can represent the code more accurately and above all allow for more types of analysis making it more useful.

7 Appendix

7.1 Source code

The full source code can be found here:

https://github.com/Tokoen/solidity-air

7.2 Analysis repositories

Here is the full list of repositories that have been used for the analysis:

1. https://github.com/OpenZeppelin/openzeppelin-contracts
2. https://github.com/aave/aave-v3-core
3. https://github.com/Sol-DAO/solbase
4. https://github.com/ethereum/remix-workshops
5. https://github.com/OffchainLabs/bold
6. https://github.com/lens-protocol/core
7. https://github.com/ProjectOpenSea/seaport
8. https://github.com/AFKDAO/ERC4610
9. https://github.com/FraxFinance/frax-solidity
10. https://github.com/enzymefinance/protocol
11. https://github.com/beefyfinance/beefy-contracts
12. https://github.com/yieldprotocol/vault-v2
13. https://github.com/worldcoin/world-id-contracts
14. https://github.com/sentimentxyz/protocol
15. https://github.com/pendle-finance/pendle-core-v2-public
16. https://github.com/Relic-Protocol/relic-contracts
17. https://github.com/AngleProtocol/angle-transmuter
18. https://github.com/wildcat-finance/wildcat-protocol
19. https://github.com/teller-protocol/teller-protocol-v2
20. https://github.com/storyprotocol/protocol-contracts

References

[1] Contracts — solidity 0.8.27 documentation. https://docs.soliditylang.org/en/latest/contracts.html, 2024.

[2] Cwi - software analysis and transformation. https://github.com/cwi-swat, 2024.

[3] Language grammar — solidity 0.8.27 documentation. https://docs.soliditylang.org/en/latest/grammar.html, 2024.

[4] module analysis::m3::ast — the rascal meta programming language. https://www.rascal-mpl.org/docs/Library/analysis/m3/AST/, 2024.

[5] module analysis::m3::core — the rascal meta programming language. https://www.rascal-mpl.org/docs/Library/analysis/m3/Core/, 2024.

[6] module lang::java::m3::core — the rascal meta programming language. https://www.rascal-mpl.org/docs/Library/lang/java/m3/Core/, 2024.

[7] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming*, 79:241–259, 2014. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

[8] Bas Basten, Mark Hills, Paul Klint, Davy Landman, Ashim Shahi, Michael Steindorfer, and Jurgen Vinju. M3: A general model for source code analytics in rascal. In *Proceedings of International Workshop on Software Analytics 2015 (SWAN 2015)*, March 2015.

[9] Monika di Angelo, Thomas Durieux, João F. Ferreira, and Gernot Salzer. Smartbugs 2.0: An execution framework for weakness detection in ethereum smart contracts. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 2102–2105, 2023.

[10] Stéphane Ducasse, Nicolas Anquetil, Muhammad Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. Research report, November 2011.

[11] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15, 2019.

[12] Paul Klint, Tijs van der Storm, and Jurgen Vinju. *EASY Meta-programming with Rascal*, pages 222–289. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[13] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal, 10 years later. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 139–139, 2019.

[14] Chao Peng, Sefa Akca, and Ajitha Rajan. Sif: A framework for solidity contract instrumentation and analysis. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–473, 2019.

[15] Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, and Ilias Tsatiris. Symbolic value-flow static analysis: deep, precise, complete modeling of ethereum smart contracts. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[16] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '18, page 9–16, New York, NY, USA, 2018. Association for Computing Machinery.

[17] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 67–82, New York, NY, USA, 2018. Association for Computing Machinery.