2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS) | 978-1-6654-3924-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/BRAINS52497.2021.9569799

An Automated Framework for Migrating Java Applications to Ethereum Solidity Applications

Akshay M. Fajge, Siddharth Thakur, Rahul Kumar and Raju Halder
Indian Institute of Technology Patna, Patna, India
Email: {fajge_1921cs12, siddharth.cs16, rahulkumar.cs17, halder}@iitp.ac.in

***Abstract*—With the advancement of blockchain technology at a breakneck pace, numerous organizations are now experimenting with blockchain solutions for their operations. In this nascent stage, due to the scarcity of blockchain domain experts, there is a growing trend towards an automated conversion of legacy systems to blockchain-based systems. To this aim, this paper presents an automated framework that facilitates the migration of centralized Java-based applications to decentralized Solidity-based Ethereum applications. The experimental results are encouraging, demonstrating its ability to handle large-scale Java codebase. To the best of our knowledge, this is the first proposal of its kind that enables the translation of Java source codes to Solidity source codes.**

***Index Terms*—Blockchain, Ethereum, Smart Contract, Java, Solidity**

I. Introduction

In recent years, the notable success of blockchain technology has piqued the interest of both industry and academia. With its advancement at a breakneck pace, numerous organizations are now extensively involved in experimenting with blockchain solutions for their operations [1].

While Bitcoin's [2] popularity has boosted the widespread adoption of blockchain technology particularly in the financial sector, Ethereum [3] has emerged as a second-largest public blockchain network and a most popular blockchain platform for distributed applications. This is due to its ability to execute Turing Complete programs (known as smart contracts) on a decentralized network through its execution engine called the Ethereum Virtual Machine (EVM). To this aim, Ethereum development community designed Solidity programming language with a complete support to blockchain-specific features [4]. Solidity has evolved into the most actively developed and maintained language on the Ethereum blockchain, making it the obvious choice for Ethereum developers when writing smart contracts. Although few blockchain platforms support mainstream languages, such as Java [5], C++ [6], GoLang [7], etc., these languages are proven to be unfit in many situations due to the presence of non-determinism and language vulnerabilities [7], [8].

There is growing trend toward converting legacy systems to blockchain-based systems. However, transitioning from existing technology to blockchain technology is a challenge, and generally, industries perform upgrading using one of the following ways:

- *Develop the blockchain solution from scratch.* While this is often the only reasonable course of action, but it is frequently a most expensive due to the time and money invested. It also requires a dedicated team of developers with domain expertise in target languages.
- *Migrate the existing codebase to blockchain solution.* In comparison to the first approach, this one necessitates the establishment of a dedicated team of developers with domain expertise in both source and target languages. This approach, however, is less expensive and time-consuming than the first, as it only requires rewriting the existing codebase in the target language.
- *Adopt any similar off-the-shelf blockchain solution.* It requires extensive customization and can become quite costly. Additionally, organizations must rely on a third party to modify their solution over time.

The first two approaches require developing robust, secure, and efficient DApps by developers. However, a scarcity of blockchain domain-experts during the development phase increases transition cost, primarily because of the new software stack. Instead, when a source-to-source translator is used in conjunction with the assistance of DApp developers, migration is unquestionably more cost-effective, fast, accurate, and bug-free than starting from scratch or migrating without the assistance of an auto-translator. Considering Java as a mostly adopted programming language[1], this paper presents an automated framework which facilitates the migration of Java-based applications from a centralized setting to a blockchain-oriented decentralized setting. To summarize, this paper contributions the followings:

- We propose the architecture of an automated source code translator from Java to Solidity. This approach performs the translation at the abstract syntax tree (AST) level.
- We develop a working prototype of a Java-to-Solidity Translator. Additionally, this package includes a Solidity AST to Solidity code generation tool.
- We conduct extensive testing on various large-scale standard Java programs and provide statistics and analytics on Java code parsing time, AST-to-AST translation time, Solidity code generation time, and memory usage.

To the best of our knowledge, this is the first proposal of its kind that enables the translation of Java source code to Solidity source code.

978-1-6654-3924-4/21/$31.00 ©2021 IEEE

[1]https://www.tiobee-index

TABLE I
PERFORMANCE EVALUATION OF JAVA-TO-SOLIDITY TRANSLATOR

Input File	Java Source Code				Solidity AST Generation Time *(sec)*	Solidity Code Generation Time *(sec)*	Memory Usage *(MB)*	Solidity *#LOC*	Failure %
	#Classes	*#Methods*	*#LOC*	*Parsing Time (sec)*					
BigDecimal.java	4	149	4933	3.18	8.22	5.99	47.54	4747	0.03
Integer.java	2	51	1497	1.20	2.15	1.23	32.31	1478	0.01
BigInteger.java	2	123	3916	2.14	6.54	4.67	120.77	3290	0.02
Character.java	4	99	6704	2.61	7.11	2.88	41.77	4470	0.02
Math.java	2	75	2256	1.31	2.37	1.02	19.71	2109	0.02

Read as number.

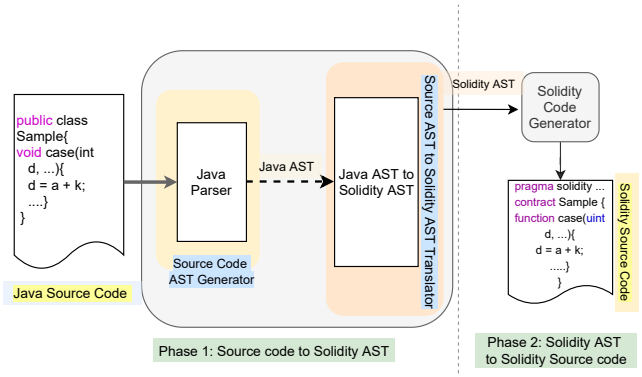

Fig. 1. Architecture of the Proposed System

II. THE PROPOSED SYSTEM: JAVA-TO-SOLIDITY

This section describes our proposed system for converting programs written in legacy languages to Solidity while maintaining their specifications. The proposed system's architecture is depicted in Figure 1, which consists of the following phases:

- Phase 1: *Source code to Solidity AST generation*
- Phase 2: *Solidity AST to Solidity code generation*

Now, let us describe each phase in detail.

A. Source code to Solidity AST generation

This phase consists of two functional components. The first one checks the syntactic correctness of given source programs using source language-specific parser and generates their corresponding AST using source code AST generator module. The other one performs post-order traversal of the source code AST and generates its equivalent Solidity AST recursively using the mapping library included in the Source AST to Solidity AST translator module. The mapping file defines which node in the source AST corresponds to which node in the target AST. The information extracted from the nodes of the source AST is inserted into the nodes of the Solidity AST in accordance with language's specifications. We generate the AST of Java source programs using the open-source Java parser core library developed by JavaParser[2] organization. Table II summarises several significant mappings of Java AST nodes to Solidity AST nodes.

[2]http://javaparser.org/

TABLE II
JAVA TO SOLIDITY AST NODES MAPPING.

Java AST node	Solidity AST node
CompilationUnit	SourceUnit
ClassOrInterfaceDeclaration	ContractDefinition
MethodDeclaration	FunctionDefinition
BlockStmt	Block
Parameter	VariableDeclaration
FieldDeclaration	VariableDeclaration
ReturnStmt	Return
VariableDeclarationExpr	VariableDeclarationStmt
VariableDeclarator	VariableDeclaration
PrimitiveType	ElementaryTypeName
ArrayType	ArrayTypeName
NameExpr	Identifier
ForStmt	ForStatement
MethodCallExpr	FunctionCall
ArrayAccessExpr	IndexAccess
ObjectCreationExpr	FunctionCall

B. Solidity AST to Solidity code generation

In contrast to the previous phase, this one is not dependent on the source code. The Solidity code generator has a node processing method for each node in the Solidity AST. It generates Solidity source code from the specification of the Solidity AST. It traverses the Solidity AST and uses a functional interface to generate Solidity source code, which is then recursively embedded in the appropriate location in the source code file.

III. EXPERIMENTAL RESULTS AND ANALYSIS

We compiled a list of differences between Solidity and Java and identified the changes required during the translation. We addressed several design challenges, such as for-each loop, switch case, constructor overloading, and static method, since Solidity lacks support for these Java features. We achieved a 0% error rate for the language building components that are common in Java and Solidity. To demonstrate the tool's applicability, we conducted an experiment to evaluate the tool's performance employing Standard Java libraries. Table I contains a representative sample of the performance evaluation result of Java-to-Solidity translator. For the given input Java source code (shown in column 1), columns 2-4 represent the number of classes, class methods, and lines in it. The time taken by the parser to parse the Java code in order to generate

its AST is reported in column 5. Columns 6-9 report the time required to generate Solidity AST from Java AST, the time required for translating Solidity AST to Solidity source code, the memory usage, and the percentage of translation failure. We define the translation failure as the ratio of the failed AST nodes in the translation to the total number of nodes in the source AST. All experiments were conducted on a Windows platform equipped with a 1.9 GHz Intel Core i3 processor and 4 GB of RAM. Observably, the result of our experimental evaluation is encouraging and it demonstrates our prototype's effectiveness in managing large Java codebase with minimal time and memory consumption.

IV. Conclusion and Future Plans

This paper proposes an architecture of an automated translator to Solidity language. We develop a working prototype for Java-to-Solidity translation, which is capable of translating either a single Java file or an entire Java codebase in a single run. Our proposal can easily be extended to other programming languages or specifications by simply adding a source-specific parser module and a Solidity AST translator module in phase 1. In the future, we plan to conduct safety and security analyses while generating the Solidity source code. We also plan to analyze the semantic equivalence of the source language programs and the translated Solidity code.

Acknowledgment

First author's research work is supported by Visvesvaraya PhD Scheme, MeitY, Govt. of India (MEITY-PHD-3009).

References

[1] F. Casino, T. K. Dasaklis, and C. Patsakis, "A systematic literature review of blockchain-based applications: Current status, classification and open issues," *Telematics and Informatics*, vol. 36, pp. 55–81, 2019.

[2] S. Nakamoto. (2008) Bitcoin: A peer-to-peer electronic cash system. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[3] G. Wood. (2014) Ethereum: A secure decentralised generalised transaction ledger. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[4] Ethereum. Solidity documentation – release 0.8.4. [Online]. Available: http://solidity.readthedocs.io/

[5] F. Spoto, "A java framework for smart contracts," in *Proceedings of the Financial Cryptography and Data Security Workshop on Trusted Smart Contracts (WTSC'19)*. Frigate Bay, St. Kitts and Nevis: Springer LNCS 11599, February 18-22 2019, pp. 122–137.

[6] EOSIO. Eosio official portal. [Online]. Available: https://eos.io/

[7] E. Androulaki *et al.*, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference (EuroSys'18)*. Porto, Portugal: ACM, April 23-26 2018, pp. 30:1–30:15.

[8] F. Spoto, "Enforcing determinism of java smart contracts," in *Proceedings of the Financial Cryptography and Data Security Workshop on Trusted Smart Contracts (WTSC'20)*. Kota Kinabalu, Malaysia: Springer LNCS 12063, February 14 2020, pp. 568–583.