Jabuti CE: A Tool for Specifying Smart Contracts in the Domain of Enterprise Application Integration

Mailson Teles-Borges[1][a], Jose Bocanegra[2][b], Eldair F. Dornelles[1][c], Sandro Sawicki[1][d], Antonia M. Reina-Quintero[3][e], Carlos Molina-Jimenez[4][f], Fabricia Roos-Frantz[1][g] and Rafael Z. Frantz[1][h]

[1] *Unijui University, Ijuí, RS, Brazil*

[2] *Universidad de los Andes, Bogotá, Colombia*

[3] *University of Seville, Seville, Spain*

[4] *Department of Computer Science and Technology, University of Cambridge, U.K.*

Keywords: Blockchain, Domain-Specific Languages, Enterprise Application Integration, Jabuti DSL, Language Server Protocol, Smart Contracts.

Abstract: Some decentralised applications (such as blockchains) take advantage of the services that smart contracts provide. Currently, each blockchain platform is tightly coupled to a particular contract language; for example, Ethereum supports Serpent and Solidity, while Hyperledger prefers Go. To ease contract reuse, contracts can be specified in platform-independent languages and automatically translated into the languages of the target platforms. With this approach, the task is reduced to the specification of the contract in the language statements. This can be tedious and error-prone unless the language is accompanied by supportive tools. This paper presents Jabuti CE, a model-driven tool that assists users of Jabuti DSL in specifying platform-independent contracts for Enterprise Application Integration. We have implemented Jabuti CE as an extension for Visual Studio Code.

1 INTRODUCTION

A smart contract is a software artefact designed to verify and execute transactions. It is normally written in a contract language such as Solidity and Go, and is deployed on a blockchain (Khan et al., 2021). Usually, software engineers write a contract manually in the run-time language; however, this approach suffers from two serious drawbacks: firstly, the contract is not reusable because platforms are tightly coupled to a particular contract language, for example, Ethereum supports Serpent and Solidity while Hyperledger prefers Go; secondly, the contract is likely to be buggy and vulnerable (Durieux et al., 2020) because current contract languages are not easy or intuitive to use; and they include too many language details (e.g., variables, pointers, data structures) that distract the programmer from the semantics of the contract. To avoid these problems, the programmer can use programming environments that include tools to specify a contract in a platform-independent language and to automatically translate it into the language of the target platform. The implementation of these tools requires a great deal of programming skills; therefore, they are not widely available.

To cover this gap, the main contribution of this article is the Jabuti DSL Contract Editor (Jabuti CE) [1]. We have implemented it as an extension of Visual Studio Code (VSCode) to assist Jabuti DSL programmers at contract editing time. Jabuti DSL (Dornelles et al., 2022) is a Domain Specific Language for writing

[a] https://orcid.org/0000-0001-7674-854X
[b] https://orcid.org/0000-0002-8342-7346
[c] https://orcid.org/0000-0001-6585-3432
[d] https://orcid.org/0000-0002-7960-0775
[e] https://orcid.org/0000-0003-3698-6302
[f] https://orcid.org/0000-0002-3617-8287
[g] https://orcid.org/0000-0001-9514-6560
[h] https://orcid.org/0000-0003-3740-7560

[1] More details of the tool and the Appendix to this article are available at https://github.com/gca-research-group/jabuti-dsl-language-vscode

Teles-Borges, M., Bocanegra, J., Dornelles, E., Sawicki, S., Reina-Quintero, A., Molina-Jimenez, C., Roos-Frantz, F. and Frantz, R.
Jabuti CE: A Tool for Specifying Smart Contracts in the Domain of Enterprise Application Integration.
DOI: 10.5220/0012413300003645
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2024)*, pages 195-202
ISBN: 978-989-758-682-8; ISSN: 2184-4348
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

platform-independent contracts for enterprise application integration (EAI). Among other features, Jabuti CE can detect syntax errors and suggest solutions.

The paper is structured as follows: in Section 2 we present the core concepts; Section 3 describes the tool development flow; related work is discussed in Section 4; finally, in Section 5 we present results and outline ongoing and future work.

2 BACKGROUND

In this section, we will first explain how the data exchange process works in the context of Enterprise Application Integration, and then we will present the Jabuti constructors.

2.1 Enterprise Application Integration

Enterprise Application Integration (EAI) is the branch of software engineering that deals with the integration process that allows different applications to exchange information with each other (Soomro and Awan, 2012). Communication between these applications and the integration process occurs through ports, which are implementations of protocols such as smtp, sftp, http, amqp, among others. These ports can be unidirectional or bidirectional (Dornelles et al., 2022; Parahyba et al., 2022). When the port is unidirectional, messages flow in a single direction. Unidirectional ports can be classified into two categories: *Entry* and *Exit*.

Entry. It is the data entry port in the integration process and allows two operations:

- Push. Adds application data to the integration process;
- Read. Reads data from the integration process and adds it to the application.

Exit. It is the data exit port in the integration process and allows two operations:

- Poll. Reads data provided by the integration process;
- Write. Writes data in the integration process.

In bidirectional ports, the flow of messages travels in two opposite directions through two types of ports: *Solicitor* and *Responder*. These ports have only two types of operations *request* and *response*. Ports of type *Solicitor* perform operations of type *request* allowing the application to request data or services from the integration process. Ports of type *Responder* perform operations of type *response* that return the requested data.

2.2 Jabuti DSL

In the context of EAI, the rules of the communication contract (or agreement) through the ports between the application and the integration process can be represented using smart contracts. Smart contracts are self-executing digital representations of traditional contracts, defining rights, obligations, prohibitions, and penalties for the parties involved (Dornelles et al., 2022).

Jabuti is a model-driven language for smart contracts in the field of EAI that helps to write platform–independent contracts. These contracts can be translated automatically into general-purpose (e.g. Java, C, Go) or cross-domain languages (e.g. Solidity) that are specific to blockchains. Jabuti comprises the following concepts:

Contract. It is the main constructor and defines the scope of the contract. It defines the two participants, the start and end dates, and one or more clauses, terms, and events.

Dates. Defines the validity period of the contract with the start and end dates.

Party. Represents the two contracting parties, namely, the integration process and the application under integration.

Clause. Specifies the responsible party (*rolePlayer*) and the conditions (*terms*) for executing an operation. Operactions can be *push*, *read*, *poll*, *write*, *request*, and *response*. There are 3 types of clauses:

- Right. The *rolePlayer* has the right to execute the clause or not.
- Obligation. The execution of the clause is mandatory.
- Prohibition. Specifies clauses that prohibit contract execution.

Terms. defines the service-level rules and business rules that must be met by the contracting parties.

To help to understand our approach better, we present a real scenario in which integration rules can be represented using Jabuti DSL. In the scenario an e-commerce hires a payment service that will be responsible for processing payments for orders created by consumers. The e-commerce will request a new transaction from the payment service that receives and will return a corresponding identifier in response. Then, the payment service will send notifications to the e-commerce informing it of status changes that have occurred.

Figure 1 illustrates this process and some rules stipulated through a representation via BPMN. We

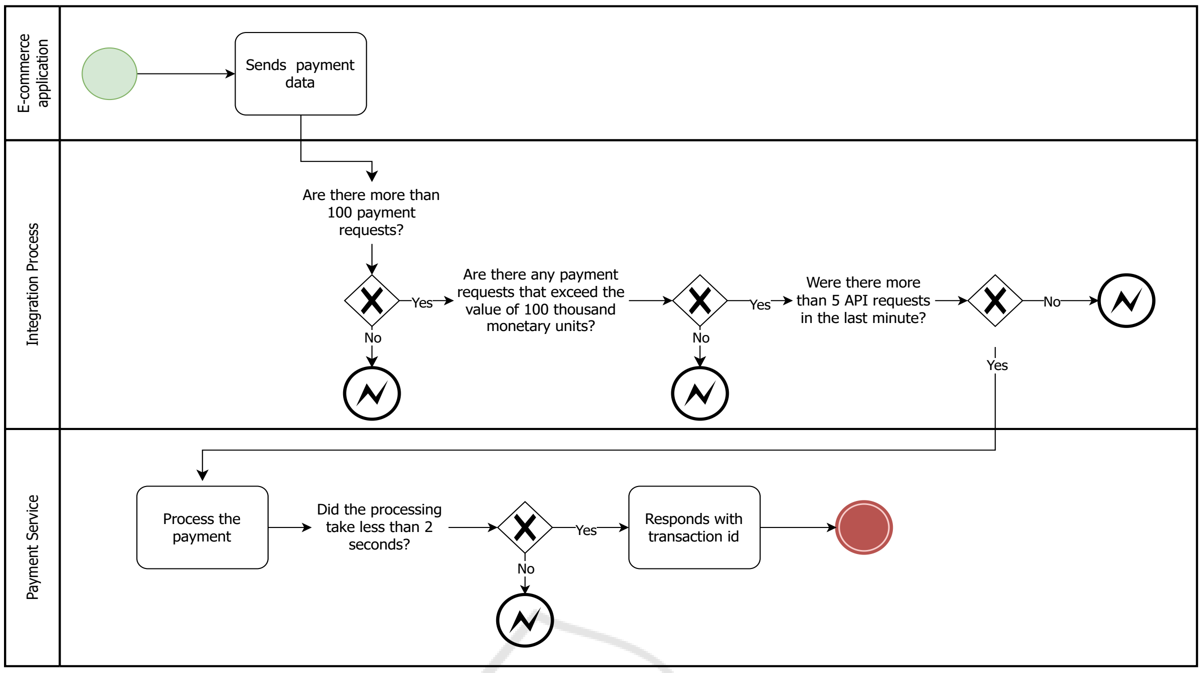

E-commerce application
Integration Process
Payment Service
Sends payment data
Are there more than 100 payment requests?
Yes
No
Are there any payment requests that exceed the value of 100 thousand monetary units?
Yes
No
Were there more than 5 API requests in the last minute?
No
Yes
Process the payment
Did the processing take less than 2 seconds?
Yes
No
Responds with transaction id

Figure 1: BPMN diagram of the integration process between e-commerce and payment service.

can observe the rules that establish the duties of each party involved: the e-commerce application can access the integration process a maximum of 5 times per minute; each request may contain a maximum of 100 transactions; each transaction will have a maximum value of one hundred thousand monetary units; the payments service will have up to 2 seconds to process and return a response. Listing 1 shows these rules written with Jabuti DSL.

Although Jabuti DSL has an optimised syntax for writing smart contracts, using it without the support of an editor or a tool that can validate the written contract, provide suggestions, or differentiate the constructors present is a challenge.

3 TOOL OVERVIEW

In the implementation of Jabuti CE, we have opted for extending VSCode, an emerging technology that has some advanced features such as language support, user interface, extension ecosystem, and performance. The implementation of Jabuti CE involves the steps shown in Figure 2: (i) grammar specification; (ii) implementation of the editor's functionalities; (iii) development of plugin for communication with the editor; (iv) building and deployment of the extension in the VSCode marketplace. We detail these steps in the subsequent subsections.

3.1 Specifying the Grammar

For the selection of the library responsible for specifying the grammar, we have taken into account three criteria: low learning curve, maintainability, and availability of development support tools. A low learning curve requires both the availability of documentation and the structure of the grammar language (e.g. if it is simple enough to be understood). Maintainability means simple procedures to upgrade, identify, and fix bugs. Finally, the availability of development support tools implies the existence of plugins or editors to write the grammar and tools to identify errors during the development stage.

We have chosen the ANTLR library (Parr, 2023) because it meets the above requirements. For instance, it has plugins for the main editors or IDEs; the generated code is human-readable; it is well documented and provides examples; in addition, it includes capabilities to convert the grammar into other languages (e.g. Typescript, Java, C#, Php among others).

In Listing 2 we provide an excerpt of the specification of the grammar of Jabuti DSL [2]. Lines 4 to 8 represent the token declaration (the smallest semantic units of the language); Line 13 shows the rule declaration for *contract*. Note that tokens are written in

[2]The entire grammar of the language can be seen at https://github.com/gca-research-group/jabuti-dsl-language-vscode

```
contract PaymentService {
  dates {
    beginDate = 2023-09-25 21:21:42
    dueDate = 2023-09-26 21:21:42
  }

  parties {
    application = "E-commerce"
    process = "Payment Service"
  }

  variables {
    payments = "$.data.length()"
    value = "$.data.[*].value"
  }

  clauses {
    right requestPayment {
      rolePlayer = application
      operation = request

      terms {
        MaxNumberOfOperation(5 per
          ↪ Minute),
        MessageContent(payments <= 100)
          ↪ ,
        MessageContent(value <= 100000)
      }
    }

    obligation responsePayment {
      rolePlayer = process
      operation = response

      terms {
        Timeout(2)
      }
    }
  }
}
```

Listing 1: Contract for the payment integration process written in Jabuti DSL.

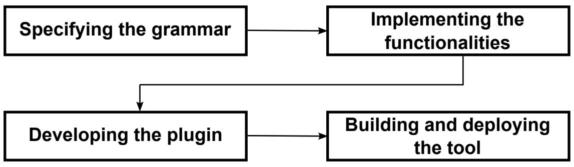

Specifying the grammar
Implementing the functionalities
Developing the plugin
Building and deploying the tool

Figure 2: Tool development flow.

Pascal case and non-terminal symbols in Camel case. The colon (:) at line 14 is a separator that separates the rule name from its implementation. Lines 15-19 show the expression that represents the contract.

A contract starts with the keyword *Contract* followed by its name, which is controlled by the *variableName* rule. A variable name must start with a letter followed by zero or more letters and numbers. The name of the contract is followed by the *OpenBrace* token ({). The rules are then specified for variables, dates, parties, and clauses. These rules are enclosed in parentheses and separated by *pipe* (|) to indicate that any of them can be inserted after *OpenBrace* in no particular order. Finally, at line 19, the declaration ends with the token *CloseBrace* (}) and a question mark (?). The '?' symbol represents that the foregoing expression is optional; therefore, an empty Jabuti DSL file will be taken as valid.

```
1  grammar JabutiGrammar;
2
3  // Tokens
4  Contract: 'contract';
5  Dates: 'dates';
6  BeginDate: 'beginDate';
7  DueDate: 'dueDate';
8  Parties: 'parties';
9
10 ...
11
12 // Lexer
13 contract
14     :
15     (
16         Contract variableName OpenBrace
17         (variables|dates|parties|clauses)*
18         CloseBrace
19     )?
20     ;
21
22 clauses
23     :
24     Clauses
25     OpenBrace
26     (right|prohibition|obligation)+
27     CloseBrace
28     ;
29 ...
```

Listing 2: Excerpt of Jabuti DSL grammar.

3.2 Language Server Protocol

For the implementation of the editor's functionalities (code completion, error validation, outline definition, among others), we have selected the Language Server Protocol (LSP) (Microsoft, 2023), a client-server protocol developed by Microsoft in 2016. As IDEs support different languages, LSP emerged as a way to separate and decouple the functionality of these editors, allowing the reuse of the implementation in different places.

We take advantage of three LSP features: versatility, interoperability, and bidirectional communication (Bünder, 2019). Firstly, versatility allows the server to be implemented in any language. Secondly, interoperability allows a single implementation to serve several editors/IDEs as long as they support the protocol. Finally, bidirectional communication enables the editor/IDE to send information to the server, and vice versa. From the LSP specification, it is worth highlighting the following features:

1. Code completion: identifies the cursor position and suggests what the next typed token would be.
2. Goto definition: when selecting a keyword, it allows navigation to the origin of this keyword, allowing the user to easily obtain its definition.
3. Symbol definition: enables the development of a tree for quick access to method and variable definitions in the editor.
4. Hover: add descriptive text to language keywords.

As a protocol, LSP only defines a specification of how communication between both the client and the server should occur, but without providing an implementation. It can be done in any language that includes the libraries needed. As Jabuti CE was developed in TypeScript, we chose *vscode-languageserver*, a library provided by Microsoft (Microsoft, 2023) that is also written in TypeScript. This library provides a simplified API that facilitates server implementation with methods such as *onCompletion*, *onHover*, *onDocumentSymbol* and *onDefinition*. These methods are abstractions that represent the features defined in the protocol.

3.3 Plugin

The communication between VSCode and the LSP server requires the development of a plugin. This plugin can be developed with Yeoman (Yeoman, 2023) (a tool for scaffolding apps), which creates a minimal tool structure. In addition to communication with the server, Yeoman also adds some extra features to the language, such as syntax highlighting.

For communication with the LSP server, we need to define the connection parameters. As this server is also written in TypeScript, it will be transpiled into Javascript and packaged together with the extension. This structure means that we do not need a physical server to host the application and we do not need to develop monitoring systems for this server. To define syntax highlighting, it is possible to use TextMate (Macromates, 2023), an editor for macOS, whose grammar is recognised by VSCode.

3.4 Deploy

To publish Jabuti CE on the VSCode marketplace, three steps are required: transpilation, compression, and publication.

The transpilation step is necessary because both the extension and the LSP server are written in TypeScript, and both need to be converted to JavaScript code. Although this task can be performed through TypeScript itself, this action is not recommended since the size of the final tool tends to be very large. The Microsoft recommendation is to use a packaging tool such as *esbuild* or *webpack*. We selected *esbuild* as it requires fewer configurations. We detected that when Jabuti CE was compiled with pure TypeScript, the final size of the tool was approximately 15MB. On the contrary, *esbuild* produces a tool of approximately 800KB only.

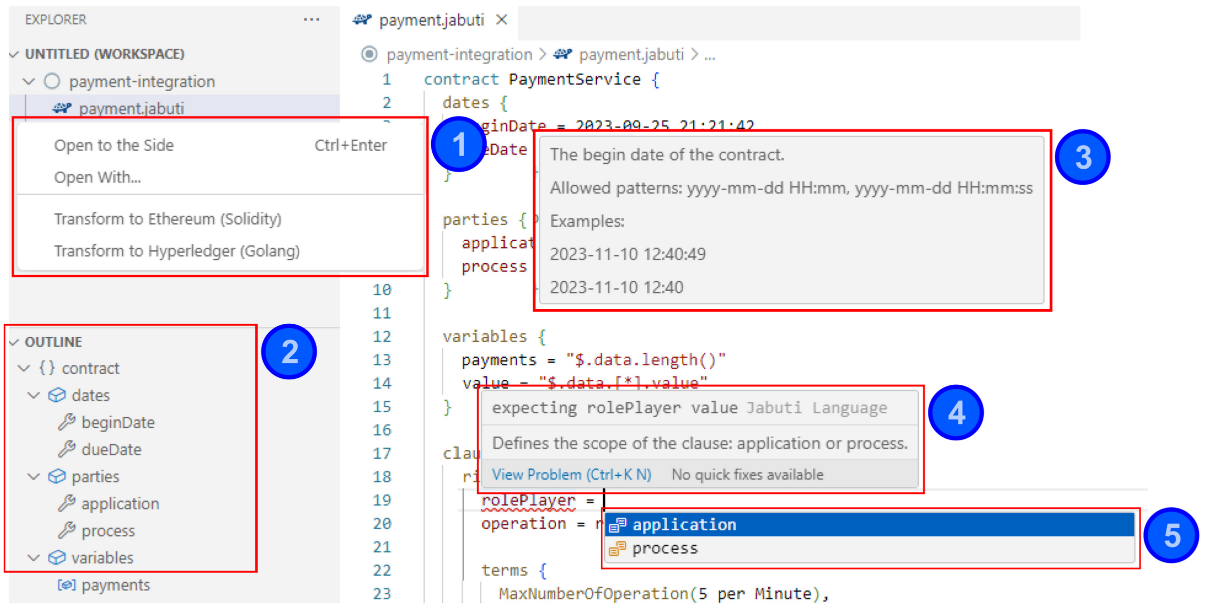
Compression is the process of packing the generated files in the format used by VSCode, which is '.vsix'. This can be done through the Visual Studio Code Extension Manager (vsce), a command line interface for compressing, managing, and publishing extensions. With the compression process, the extension for Jabuti CE was even smaller, 300KB only. Finally, the publication takes place manually by installing the generated .vsix file, or from the vsce cli, or through the administrative panel available in Azure, where the marketplace is hosted[3].

3.5 Editor

Figure 3 shows some of the features of the tool:

1. Label 1 shows the contextual menu that is accessed by right-clicking on the .jabuti file and in it we can see the two options for transformation: *Transform to Ethereum (Solidity)* and *Transform to Hyperledger (Golang)*.
2. Label 2 shows the left side menu called Outline that has a tree with the attributes and functions present in the contract under edition.
3. Label 3 highlights the description provided to users when the mouse hovers over a keyword such as *contract*, *variable*, or *date*. In addition to the description, if the word represents a block or function, an implementation example is displayed, or if the word represents an attribute, the possible values are displayed.
4. Label 4 shows how the errors are displayed in the editor. In this concrete example, we can see how

[3]The published extension can be downloaded from https://marketplace.visualstudio.com/items?itemName=gca-unijui.jabuti-language



```
contract PaymentService {
  dates {
    ginDate = 2023-09-25 21:21:42
    eDate
  }

  parties {
    applicat
    process
  }

  variables {
    payments = "$.data.length()"
    value = "$.data.[*].value"
  }

  clau
    ri
      rolePlayer =
      operation = r

      terms {
        MaxNumberOfOperation(5 per Minute),
```

The begin date of the contract.

Allowed patterns: yyyy-mm-dd HH:mm, yyyy-mm-dd HH:mm:ss

Examples:

2023-11-10 12:40:49

2023-11-10 12:40

expecting rolePlayer value Jabuti Language

Defines the scope of the clause: application or process.

Figure 3: Jabuti CE environment.

rolePlayer is underlined by a red line. Note also that a hover over the error displays a message describing the possible cause.

5. Label 5 shows an example of the completion suggestions that are activated when the user presses the CTRL and Space keys simultaneously.

In addition to the features mentioned above, Jabuti CE provides other ones:

1. When the user presses CTRL and space to start a new contract in an empty file, two different suggestions are given to complete the description of the contract: one of them contains a simplified version of a Jabuti DSL contract and the other contains a more complete version.
2. If a semantic mistake is made, such as adding two beginDate attributes, an error is displayed, underlining the additional attribute in red.

3.6 Transformation

Transformation involves the translation of the Jabuti DSL code to the target language (e.g. Solidity, Node.JS, Go, among others). Initially, we will provide the transformation option for both the Ethereum blockchain through Solidity, as well as Hyperledger with Golang. The first step is to map the DSL data structures to the target language: *variables*, *dates*, and *parties* will be mapped to private attributes, *clauses* will be mapped to methods, and the *terms* of these clauses will be mapped to conditions in the respective methods.

Next, we execute the following steps: (i) development of the template for the target language as shown in Listing 3; (ii) conversion of the selected contract into an Abstract Syntax Tree (AST); (iii) and execution of the conversion method so that the AST is submitted to the template, replacing the template variables with the AST values. This action occurs through ejs (Ejs, 2023), a template engine for Javascript.

The implementation of the transformation is still in progress; however, a first version is already available in the extension. To access it, right-click the .jabuti file and then click on one of the following two options in the contextual menu: *Transform to Ethereum (Solidity)* or *Transform to Hyperledger (Golang)*.

4 RELATED WORK

The works analyzed in this section present DSLs or methods for representing smart contracts through more abstract structures. They also provide some means of transforming these structures into executable languages on blockchains such as Solidity. However, we can observe that there is no work that meets the specifications aimed at EAI.

iContractML (Hamdaqa et al., 2020), CML (Wohrer and Zdun, 2020) and SPESC (Chen et al., 2021; He et al., 2018) are DSLs that provide support for modelling and writing smart contracts on multiple blockchain platforms in a high-level and abstract syntax. iContractML and SPESC propose

```
export const SOLIDITY_TEMPLATE = `// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract <%= contractName %> {
   // Create the attributes
   <% variables.forEach(variable => { %>
      \n\tstring public <%= variable.name %>;
   <%}) %>

   constructor() {
      // Initialize the attributes
      <% variables.forEach((variable, index) => { %>
         \n\t\t<%= variable.name %> = "<%= variable.value %>";
      <% }) %>
   }

   // others implementations
}
`;
```

Listing 3: Excerpt of Solidity template.

a tool that handles both writing and transforming for the target blockchain. On the other hand, CML presents a specification for writing and transforming contracts.

Marlowe (Lamela Seijas et al., 2020) is a DSL for specifying financial contracts. Although its language can be translated to different blockchains, the main focus is on Cardano. The tool includes a web-based editor with a rich development environment that allows users to write contracts in Haskell (Haskell, 2023), Blockly (Blockly, 2023) and other languages.

Frantz and Nowostawski (2016) present a study on the process of converting human-readable information, rules, regulations, and laws into machine-readable code structures. Their work is based on ADICO (Attributes, Deontic, Aim, Conditions, Or Else) (Crawford and Ostrom, 1995), a theoretical model that allows understanding social institutions and their dynamics by decomposing these concepts into simple declarations of rules, for example: *People (A) must (D) vote (I) every four years (C), or else they face a fine (O)*. The authors also propose model-based transformation templates written in Scala (Scala, 2023).

5 CONCLUSIONS AND FUTURE WORK

In this paper, we have highlighted the advantages of using domain-specific languages for writing smart contracts. A DSL allows to produce platform-independent code and are accessible to professionals from the specific domain with only moderate programming skills. We argue that productivity is increased when the programmer is assisted by a programming environment that includes a powerful editing tool. To support our arguments, we have implemented Jabuti CE—an editing tool for specifying smart contracts for EAI— and released it online to expose it to public scrutiny. A quick examination will reveal its salient features; notably, it helps the programmer to discover errors and suggest solutions. In addition, it provides predefined snippets, syntax highlighting, and code explanation.

Jabuti CE is an ongoing implementation. We would like to further develop it as an end-to-end tool capable of both writing and publishing smart contracts in the EAI context. Next, we will evaluate Jabuti CE on writing and transpiling contracts and their integration and execution on different blockchains. We also plan to conduct an experiment with users to evaluate the user experience.

The current implementation of Jabuti CE is based on VSCode and, therefore, requires installation on the user's local computer. We are planning an implementation of a web version that would free the user from this burden. We would also like to explore the Monaco editor, the web-enabled editor used by VSCode.

ACKNOWLEDGEMENTS

This work was partially supported by the Coordination for the Brazilian Improvement of Higher Education Personnel (CAPES) and the Brazilian National Council for Scientific and Technological (CNPq) under the following project grants 309315/2020-4, 309425/2023-9 and 402915/2023-2 to Rafael Z.

Frantz. Antonia M. Reina Quintero has been funded by the following projects: AETHERUS (PID2020-112540RB-C44), ALBA-US (TED2021-130355B-C32) AEI/10.13039/501100011033/Unión Europea NextGenerationEU/PRTR. Carlos Molina has been funded by UKRI, grant G115169 (CAMB project). Fabricia Roos-Frantz has been funded by the Research Support Foundation of the State of Rio Grande do Sul in Brazil (FAPERGS), under grant 19/2551-0001782-0, and also by the Brazilian National Council for Scientific and Technological (CNPq), under grant 311011/2022-5.

REFERENCES

Blockly (2023). Blockly language. https://developers.google.com/blockly. Accessed: June 20, 2023.

Bünder, H. (2019). Decoupling language and editor-the impact of the language server protocol on textual domain-specific languages. In *MODELSWARD*, pages 129–140.

Chen, E., Qin, B., Zhu, Y., Song, W., Wang, S., Chu, C.-C. W., and Yau, S. S. (2021). Spesc-translator: Towards automatically smart legal contract conversion for blockchain-based auction services. *IEEE Transactions on Services Computing*, 15(5):3061–3076.

Crawford, S. E. S. and Ostrom, E. (1995). A grammar of institutions. *American Political Science Review*, 89(03):582–600.

Dornelles, E. F., Parahyba, F., Frantz, R. Z., Roos-Frantz, F., Reina-Quintero, A. M., Molina-Jiménez, C., Bocanegra, J., and Sawicki, S. (2022). Advances in a DSL to specify smart contracts for application integration processes. In *Proc. of XXV Ibero-American Conf. on Software Engineering*, pages 46–60. SBC.

Durieux, T., Ferreira, J. F., Abreu, R., and Cruz, P. (2020). Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proc. ACM/IEEE 42nd Int'l Conf. on Software*.

Ejs (2023). Embed javascript templates. https://ejs.co/. Accessed: June 20, 2023.

Frantz, C. K. and Nowostawski, M. (2016). From institutions to code: Towards automated generation of smart contracts. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 210–215. IEEE.

Hamdaqa, M., Metz, L. A. P., and Qasse, I. (2020). Icontractml: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. In *Proceedings of the 12th System Analysis and Modelling Conference*, pages 34–43.

Haskell (2023). Haskell language. https://www.haskell.org/. Accessed: June 20, 2023.

He, X., Qin, B., Zhu, Y., Chen, X., and Liu, Y. (2018). Spesc: A specification language for smart contracts. In *42nd IEEE Annual computer software and applications conf. (COMPSAC)*, volume 1, pages 132–137. IEEE.

Khan, S. N., Loukil, F., Ghedira-Guegan, C., Benkhelifa, E., and Bani-Hani, A. (2021). Blockchain smart contracts: Applications, challenges, and future trends. *Peer-to-peer Networking and Applications*, 14:2901–2925.

Lamela Seijas, P., Nemish, A., Smith, D., and Thompson, S. (2020). Marlowe: implementing and analysing financial contracts on blockchain. In *Financial Cryptography and Data Security (FC'20). Int'l Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, Feb 14, Revised Selected Papers 24*, pages 496–511. Springer.

Macromates (2023). TextMate Manual - Snippets. Webpage. Accessed: June 20, 2023.

Microsoft (2023). Language Server Protocol (LSP) Specification. Website. Accessed on June 15, 2023.

Microsoft (2023). Microsoft/vscode-languageserver-node. GitHub Repository. Accessed: June 20, 2023.

Parahyba, F., Dornelles, E. F., Roos Frantz, F., Frantz, R. Z., Molina Jiménez, C., Reina Quintero, A. M., Bocanegra, J., and Sawicki, S. (2022). On the need to use smart contracts in enterprise application integration. In *CIbSE 2022: XXV Ibero-American Conference on Software Engineering (2022), pp. 203-217.* Sociedade Brasileira de Computação.

Parr, T. (2023). Antlr: Another tool for language recognition. Website. Accessed on June 15, 2023.

Scala (2023). Scala language. https://www.scala-lang.org/. Accessed: June 20, 2023.

Soomro, T. R. and Awan, A. H. (2012). Challenges and future of enterprise application integration. *International Journal of Computer Applications*, 42(7):42–45.

Wohrer, M. and Zdun, U. (2020). From domain-specific language to code: Smart contracts and the application of design patterns. *IEEE Software*, 37(5):37–42.

Yeoman (2023). Yeoman. https://yeoman.io/. Accessed: June 20, 2023.