JetBrains MPS and KernelF as a basis for creation of Domain Specific Languages for Blockchain in P2P energy trading

Marija Borisov*, Goran Sladić**, Gordana Milosavljević**
* Engineering Software Lab, Belgrade, Serbia
** University of Novi Sad, Faculty of Technical Sciences, Serbia
marija.borisov@gmail.com, {sladicg, grist}@uns.ac.rs

***Abstract*—Innovations in blockchain technologies are expected to have a significant effect on many industry fields. In this paper, we explore the position of Domain Specific Languages (DSLs) in today's world, especially in emerging P2P energy trading powered by blockchain. Our motivation to research these areas is the potential of P2P trading in blockchain to revolutionize the energy sector. Existing programming languages cannot solve all the issues when working with smart contracts in energy blockchain. We elaborate usage of JetBrains MPS and KernelF as a core for developing a mini-DSL for P2P energy trading. We put forward the idea that DSLs based on KernelF can overcome shortcomings in employing existing programming languages in the given domain.**

I. INTRODUCTION

Energy sustainability is among the most critical issues of the modern age. Distributed energy enabled with blockchain technology is part of the solution which can contribute to the broader inclusion of renewable energy sources into smart grids.

Blockchain in the energy sector has been recognized as a promising but disruptive technology. Lack of regulations not only in the blockchain field but also in the emerging smart grid systems may pose a problem for the global acceptance of blockchain in the energy sector. Furthermore, the leading energy companies are less willing to participate in innovative approaches than the small and medium ones.

Blockchain enables secure, transparent, tamper-proof distributed systems that eliminate the need for centralized, third-party identity [1]. This distributed ledger technology uses consensus algorithms to achieve validation and agreement. The most prominent consensus algorithms are Proof-of-Work (PoW), Proof-of-Stake (PoS), Proof-of-Authority (PoAu), and Practical Byzantine Fault Tolerance (PBFT). As blockchain aims to be green technology, many blockchain platforms move toward more sustainable consensus algorithms such as PoS. Ethereum[1] is an example of a blockchain in which the PoS algorithm will remove PoW, and this process has already started.

P2P energy trading presents the decentralized form of the energy market. In many papers that describe P2P trading, some kind of distribution system operator (DSO), an operator or manager of the grid, is still present, but P2P trading can revolutionize the existing structure of power markets. Blockchain-enabled systems have a huge part in this process, as they give the infrastructure and the means to achieve these goals. Direct energy trading between prosumers and consumers can provide more power to involved parties to control their energy production and demand [1]. As the system operators such as DSOs now manage the network and operations in it, work on reducing their roles in the smart grids is undergoing. Good results are achieved in small microgrids and various pilot projects worldwide, such as Brooklyn Microgrid [1].

Moreover, local and community microgrids are expected to be essential in future energy systems. Because the energy in these systems is produced locally, this can cause a reduction in distribution and transmission losses. Possible problems may impose many participants, the inclusion of different energy sources, and operations of the grid such as the continuous balance of demand and supply [1].

In energy trading, the concept of smart contracts is crucial. This concept is linked to Ethereum blockchains but can be transferred to Hyperledger also. Solidity[2] on Ethereum is a popular object-oriented programming language for writing smart contracts. Every smart contract has its unique address. Because they automate the execution of an agreement between the actors, including no intermediary, they present a very effective solution for P2P energy trading. This paradigm can also create new business models [3]. Nevertheless, the risks and opportunities of using smart contracts extensively must be studied in detail. Although Solidity matured over time, there are still some vulnerabilities in Solidity smart contracts, such as calls to unknown, type conversion, reentrancy, loss of ether, costly loops and gas limit, overflow, and underflow, etc. [11].

In this paper, we propose the usage of JetBrains MPS and KernelF as a core for developing a mini-DSL for P2P energy trading. We put forward the idea that DSLs based

[1] https://ethereum.org/en/upgrades/merge/

[2] https://docs.soliditylang.org/en/v0.8.14/

on KernelF can overcome shortcomings in employing existing programming languages in the given domain.

This paper is organized as follows. Section 2 reviews related work. Section 3 presents the main characteristics of a language workbench JetBrains MPS and its KernelF language. Mini-DSL for P2P energy trading is shortly described in Section 4. Section 5 concludes the paper and outlines future work.

II. Related Work

Smart contracts, in a nutshell, present programming code saved on a blockchain. A smart contract is a computer protocol that is created to verify and execute a contract on the blockchain, allowing transactions without third parties. This should ensure that they work correctly, executing the trading agreement each time the conditions are met and creating immutable records of the transaction [2]. A number of important use cases of smart contracts in the energy field are given in [1].

In today's world, in many cases, it is still impossible to enable trust between all actors to determine the price, balance the network, etc., without some third party. In [2] is given a solution that uses Ethereum blockchain and the concept of limited DSO to provide security and integrity of trading records, prevention of double sales, and automatic and autonomous network operations.

Dynamic pricing models are usually envisioned for smart grids. In [2], an adaptive and adjustable dynamic pricing scheme is proposed to balance demand and supply among prosumers and consumers. If total supply overpowers total demand, the price is reduced to the minimum. Then, the price is gradually increased until demand is much greater than supply.

In [3] is presented a state diagram for representation of the trading system. Interestingly, this property of the proposed trading system is very suitable to be modeled with JetBrains MPS state machine concept, which will be presented in this paper. The proposed DSO fully controls the created smart contract. From a technical point of view, Etherium's identity-based messaging system called Whisper is leveraged as a solution, in particular situations, for not creating traffic and skipping mining into a block.

Along with use cases in various fields, the potential of using specific DSLs in the energy sector enabled with blockchain has just started to be realized as a great way of solving many problems originating from existing programming languages application to smart contract development.

In [7] is given a short description of the DSL based on KernelF for smart contracts specification. The proposed DSLs effectively solve many of the issues known when working in Solidity.

The detailed description and specification of the KernelF language are elaborated in [8]. It should be noted that KernelF changed its reference and added constructions, so these papers are not quite up to date.

III. DSLs, JetBrains MPS, KernelF

DSL is a language optimized for a specific problem domain [4]. It has suitable syntax to describe the abstractions clearly and concisely. External DSLs are new languages built from scratch with their own syntax. Internal DSLs are embedded into an existing host language and use its syntax and development environment. Simpler and smaller DSL, for example, the one used by a single application, can be named mini-language or mini-DSL.

The line that divides DSL and GPL (General-Purpose Language) is sometimes blurred, such as in cases of Perl, PostScript, etc. It can be leveraged that domain-specificity is a gradual term. There are no clear yes or no answers in many cases. It is important to note that Solidity has many features of GPL, although it is used for smart contract creation in Ethereum.

The drawbacks of using a DSL are its potentially complex development and a small community of users, which can make new DSL development risky from an economic and business point of view [6]. Also, DSL often sacrifices flexibility to enable productivity and consistency in a given domain.

DSL's development can be optimized, and risks minimized using language workbenches such as JetBrains MPS[3] (Meta Programming System). JetBrains MPS is an open-source platform for the rapid creation of textual DSLs. DSLs are equipped with custom editors enabling code completion, system checks, static code analysis, debugging, and testing. DSL can be developed from the start as external or embedded in an existing language supported by MPS as internal DSL. It is possible to include non-textual notations, for example, mathematical formulas, tables, and graphics, in both types of DSLs.

mbeddr consists of extensible languages equipped with an IDE (Integrated Development Environment). It is primarily intended for embedded software engineering based on a variant of C programming language developed on MPS. It supports implementation, testing, verification, and process aspects. It integrates with command-line build tools, integration servers, and file-based version control systems. Its state machine extension is used in this paper for contract specification in P2P trading. State machine extension contains events based on KernelF expression language, variables, states, and transitions with guards.

KernelF is a functional language created on top of MPS which supports "funclarative programming" - a mix of functional and declarative programming. KernelF is designed to be easily used as a core of DSLs and, as such, extensible and embeddable [7]. It consists of several MPS languages that can be used independently in a newly created DSL [7]. Besides its use in different fields such as security analysis, insurance, and bookkeeping, it has started to be used for smart contracts specification. The following paragraphs provide an overview of the fundamental principles of the KernelF.

KernelF is statically typed. A handy design of KernelF regarding its usage for smart contracts is that all expressions are effect-free. Values and collections (lists, sets, maps) are immutable. Structures like boxes and state machines enable mutability when it is needed. KernelF does not have loops, just conditionals. It also does not support generics but alt expression to work with a range of numbers. "Attempt" is used for error handling.

Enumerations and records are supported. There are effect flags just for reading and for reading and modifying

[3] https://www.jetbrains.com/mps/learn/

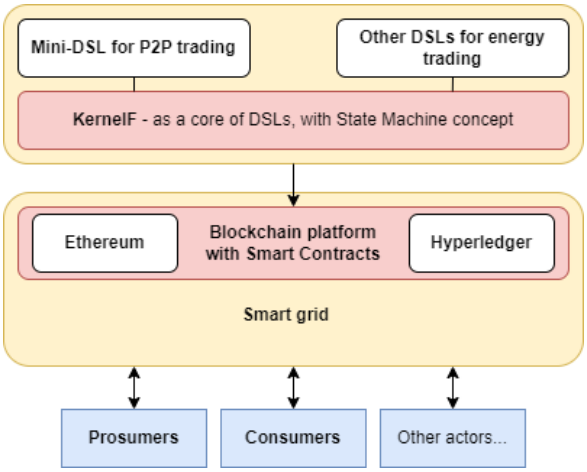

Figure 1. Mini-DSL for P2P trading architecture

that can be added to the functions. Transactions are also supported.

Interpreter, code generator, read-eval-print-loop, and debugger are present in KernelF. Creating tests that use an interpreter is very useful for checking all the values and correctness of the functions and variables in state machines, and records, used in KernelF.

IV. Mini DSL Language for Energy Trading

By examining the existing solutions for smart contracts that run on the blockchain [10], we can conclude that used GPLs and existing DSLs do not have proper support for the patterns that exist in smart contracts for the energy sector. The architecture of smart contracts is well studied, and main features are identified, which can help build appropriate DSLs. Although the blockchain guarantees the execution of a smart contract, once the conditions are met, the smart contract must be correctly implemented. KernelF is a good choice as a core language for DSL creation because it is functional, supports transactions, and its code can be easily verified. Moreover, KernelF provides low-level language constructs, making creating DSLs less challenging. Smart Contract can be viewed as a state machine, which is supported as mbedder extension [9]. Although a person without programming experience could find some difficulties in using DSLs to build smart contracts that execute energy trading logic, domain experts can easily use DSLs.

In Figure 1, the interconnections of the main aspects discussed are depicted. The smart grid is enabled with a blockchain platform on which smart contracts are executed. Created DSLs are implemented and deployed as smart contracts in the Ethereum or Hyperledger blockchain.

Based on the findings from the literature survey, especially prototypical implementations explained in [9], this paper presents a mini-DSL for energy trading on the blockchain. It is developed in JetBrains MPS and uses KernelF as a core language. This DSL implements `Offers`, `Sales`, `Accounts`, `Bids`, and `Consumers` and `Prosumers` as records. Lists of consumers and prosumers are created, so these entities are registered in the system. Functions for checking if the entity is a consumer or prosumer are given (Listing 1). Also, lists of offers and sales are created. Several important system functions are given, which can also find their place in the state machine for the contract specification. The domain expert can easily find the first prosumer, all offers by the specified prosumer, and all sales made by the consumer. Energy offer can be sold if it is not sold before that moment and is not expired. A function for a simple buying process is also implemented.

The state machine for the TradingEnergy contract initially has four states: `bids`, `finishedBids`, `toSell` and `finishedTrading`. The decision if the energy offer is going to be sold is changed in comparison to the one present in [9] because the given voting system is not suitable for energy trading.

Instead of the voting system, the concept of biding is introduced. First, consumers can give their bids on offers, and all bids are gathered. If there is at least one bid for each offer or the rest of the offers expired, the state is transferred from `bids` to `finishedBids` (Listing 2). On the other hand, if there are no bids and all offers expired, the state is changed to `finishedTranding`.

In the `finishedBids` state, the user gives a list of offers for which she wants to perform selling. The highest bid for each selected offer is calculated, and that bid is set to be accepted. Then the status transfers to the state `toSell`

In `toSell` state, for each of the highest bids, the function for buying is executed. Important aspects that

```
fun setAddress/RM(oldAddr: box<address>, addr: address) { oldAddr.update(addr) }

fun firstOfferByProsumer/R(addr: address) = offers.val.findFirst(|it.producer == addr|)
fun offersByProsumer/R(addr: address) = offers.val.where(|it.producer == addr|)

fun salesByConsumer/R(addr: address) = sales.val.where(|it.consumer == addr|)

fun canBeSold/R(offer: Energy_Offer) = offer.sold == false && offer.expiration <= now

fun buyingProcess/RM(offer: box<Energy_Offer>, price: posNum, consumer: address) {
  alt [ offer.val.sold == true ||                    => error
        offer.val.expiration > now
      offer.val.price <= price                       => {
                                                          offer.update(it.with(sold = true))
                                                          sales.
                                                              update(it.with(#Energy_Sale{1, offer.val.energy, price,)
                                                                        consumer})
                                                        }
      otherwise                                      => error
]
}

fun isConsumer/R(addr: address) = consumers.val.any(|it.account.address == addr|)
fun isProsumer/R(addr: address) = prosumers.val.any(|it.account.address == addr|)
```

Listing 1. Sample of core functions for P2P trading

```
state bids (initial) {
  on biding(offer_id, money, consumer_id, time) : {
    highest_processed.update(false)
    bids_processed.update(false)
    val chosen_offer = offers.val.where(|it.id == offer_id|)

    if chosen_offer.size > 0
      then {
      consumerBids.update(consumerBids.val.with(#Bid{offer_id, money, consumer_id, time}))
      if offers.val.id.contains(offer_id) && !existingBids.val.contains(offer_id) then
          existingBids.update(existingBids.val.with(offer_id)) else none
    }
      else none
    bids_processed.update(true)
    first.update(false)
  }
  on checkTimeout [!bids_processed.val && offers.val.all(|it.expiration < now|)] -> finishedTrading
  on bidingTransition [(offers.val.all(|{ it.expiration < now
  || existingBids.val.contains(it.id) && (first.val || bids_processed.val) }|))] -> finishedBids
}
```

Listing 2. Bids state in State machine

have an effect if an offer is sold are: (1) its expiration time and status, (2) if it is already sold or not, and (3) if the highest bid is higher than the price specified in the energy offer record. The domain expert could choose which offers to sell. Offers are updated, and in the sales list are added successfully performed trading processes.

When all offers are processed, the state machine transfers to the `finishedTrading` state. It is also chosen to allow users to perform bidding again (when they are in the `toSell` state) if all offers are not sold or expired. Then the user can go again through the states, create additional bids, choose offers to buy, etc.

Besides the definition of language, a test suite is also created to ensure the correct execution. It checks and reports the values and changes in the system. Moreover, the concept of inspectors is used, so the value of different variables can be seen after the test suite's execution. Several cases of usage and possible scenarios are given, and an assert construct is used to verify the correct execution.

Presented approach differs from the one in the traditional blockchain development based on Solidity language as follows:

- KernelF functional programming language is used as a core enabling variables to be mostly immutable (except in specific cases).
- Thanks to KernelF modularity and reusability, we created mini-DSL dedicated only to P2P energy trading. It resulted in a much simpler language that non-professional programmers can easily use.
- Data types are based on KernelF's data types, with the possibility of introducing new ones if needed.
- The State Machine concept is better suited for smart contracts description. Solidity does not have embedded support for state machines.

The code and test suits for the proposed mini-DSL can be found in [12].

V. Conclusion

The paper discusses potential opportunities of using MPS and KernelF as a base for creating DSL for energy trading on a blockchain. A blockchain-based system for P2P energy trading is described, alongside one of the dynamic pricing models. JetBrains MPS and KernelF are shortly described to present their full potential.

Usage of KernelF-based DSLs can reduce many low-level mistakes. An example of a mini-language is given as a proof-of-concept of how well KernelF-based DSLs go with smart contracts.

In future work, we plan to create a complete DSL based on the KernelF that will implement the algorithm presented in [2]. We believe that we will be able to prove that domain-specific languages can solve the problem of the implementation of P2P energy trading on blockchain efficiently and suitably. The envisioned solution will follow the presented modeling and programming approaches.

References

[1] M. Andonia, V. Robu, D. Flynn, S. Abram, D. Geach, D. Jenkins, P. McCallum, A. Peacock, Blockchain technology in the energy sector: A systematic review of challenges and opportunities, Renewable and Sustainable Energy Reviews Volume 100, February 2019, Pages 143-174, online https://www.sciencedirect.com/science/article/pii/S1364032118307184.

[2] J. G. Song, E. S. Kang, H. W. Shin, J. W. Jang, A Smart Contract-Based P2P Energy Trading System with Dynamic Pricing on Ethereum Blockchain A Smart Contract-Based P2P Energy Trading System with Dynamic Pricing on Ethereum Blockchain Sensors 2021, 21(6), 1985; doi:https://doi.org/10.3390/s21061985

[3] M. J. Bürer, M. de Lapparent, V. Pallotta, M. Capezzali, M. Carpita, Use cases for Blockchain in the Energy Industry Opportunities of emerging business models and related risks, Computers & Industrial Engineering, Volume 137, November 2019, 106002, doi: https://doi.org/10.1016/j.cie.2019.106002.

[4] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, G. Wachsmuth, DSL Engineering: Designing, Implementing and Using Domain-Specific Languages, The PDF book, 2010-2013 Markus Voelter, online http://dslbook.org

[5] mbeddr - engineering the future of embedded software, http://mbeddr.com/

[6] I. Dejanović, Prilog metodama brzog razvoja softvera na bazi proširivih jezičkih specifikacija, doktorska disertacija, FTN Novi Sad 2011.

[7] M. Voelter, The Design, Evolution, and Use of KernelF: An Extensible and Embeddable Functional Language, Conference paper, 2018, Part of the Lecture Notes in Computer Science book series (LNCS, volume 10888), doi: 10.1007/978-3-319-93317-7_1

[8] M. Voelter, KernelF - an Embeddable and Extensible Functional Language, online: http://voelter.de/data/pub/kernelf-reference.pdf

[9] M. Voelter, A Smart Contract Development Stack, online: https://languageengineering.io/a-smart-contract-development-stack-54533a3a503a

[10] A. Vacca, A. Di Sorbo, C. A.Visaggio, G. Canfora, A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges , Journal of Systems and Software, Volume 174, April 2021, 110891, doi: https://doi.org/10.1016/j.jss.2020.110891.6

[11] A. L. Vivar, A. T. Castedo, A. L. S. Orozco, L. J. G. Villalba, An Analysis of Smart Contracts Security Threats Alongside Existing Solutions, Entropy 2020, 22(2), 203, doi: https://doi.org/10.3390/e22020203.

[12] Mini-DSL for P2P energy trading source code, https://github.com/Marija-Borisov/DSL_smart_contract_energy_trading.