Context-based Role Object Pattern with On-Chain Smart Contract Programming

Orçun Oruç, Uwe Aßmann, Arbli Troshani
Technische Universität Dresden, Software Technology Group, Nöthnitzer Straße 46, 01187 Dresden

Dynamic object roles and corresponding contexts can model complex applications with higher-level abstraction. These abstracted applications can be used in wider areas such as financial institutions, health care, and supply chain network. Role management which consists of the creation of role objects, and binding role object between core objects still suffers from non-intrusive logging-monitoring, auditing, and resilient data source for role-based applications. Moreover, immutable smart contracts cause problems concerning bug fixing and maintenance without dynamic binding to new smart contract objects. An object that is created from a smart contract (contract class) can be transparently attached to a role object utilizing the Role Object Pattern (ROP). However, ROP itself does not contain a context definition and context-specific role assignment grouping the definition of smart contract relationships in abstracted data types. In this study, we would like to implement an extended version of the role object pattern called Context-based Role Object Pattern (ContextROP) with an on-chain smart contract language called Solidity to solve fundamental problems. To evaluate the proposal, we will implement a use case with the design pattern proceeding with qualitative and quantitative analysis.

Keywords: *Smart contracts, Role-based Programming, Role-Object Pattern, Object-Oriented Programming*

1. Introduction

In software engineering, a design pattern is a repeatable solution for a common design problem. One can transform a specific design pattern directly into the code. A smart contract is an autonomous entity and verifiable agreements between participants. However, a bad design of a smart contract can cause immersive problems between participants. For instance, if there is a bug in a smart contract, it must be updated on-chain network. When you deployed a new contract to update a deprecated contract, smart contract data and application logic should be altered. That is why eternal storage and upgradable contract pattern have been proposed.

Smart Contracts are deterministic, isolated, and immutable programs that can work in decentralized networks. A smart contract must give the same results under the same inputs to comply with the determinism concept. Off-chain transactions rely on a blockchain network; however, the computational logic is stored in off-chain transactions without hashed values (TransactionX). On-chain smart contracts can only provide an isolation principle because an external computation in a smart contract cannot be isolated virtual machine such as Ethereum Virtual Machine (Ethereum VM). The most important principle that we need to emphasize is the immutability of smart contracts.

Immutability issue is an important characteristic of smart contracts [2]. The immutability character of Solidity contracts can harm the credibility of auditing while changing the smart contract data structure in case of an error. Another problem could be that misused or hacked contracts, smart contracts cannot be changed [2].

„Object schizophrenia or self-schizophrenia is a combination arising from the delegation and related techniques in object-oriented programming.“[1]. Objects can have a single behavior and attribute at a specific time; however, there are some programming techniques that could not lead us to distinguish the singularity of identity. For instance, the delegation in class-based programming languages causes differen attribute personalities for an object because a delegated method can call from a base method of a base class [1]. Analogous to the class-object concept of object-oriented programming languages, smart contracts use objects from contracts (class) relationship. Contracts can be defined as class-like structures. Once a contract is deployed, the storage (data) and application logic of a contract should be deployed together. Real-world entities can be represented with roles. Entities that have several behaviors and attributes can play several roles during their lifetime. Even in object-oriented programming, each object has a set of roles because objects can be created with different properties (attributes and behaviors) from a base class. To create roles, we have a set of methods like dynamic classification, multiple classification, multiple inheritance, type hierarchy with subtyping (overriding), and subclassing. For instance, a human can be classified as *Person, Employee*, *Manager, Teacher, Student*, and *Retired Employee. Employee*, Manager, and *Retired Employee* can belong to a Factory context because they have worked in this context with attributes (name, id, social security

1 https://wiki.c2.com/?ObjectSchizophrenia

number). Some roles cannot co-exist like *Person* cannot be *Employee* and *Retired Employee* at the same time. Identity sharing can also happen in on-chain smart contracts because they use object-oriented language with the aforementioned features. This study would like to show the possible effect of identity sharing on smart contract development.

Each contract has deployment and execution asset costs concerning the operation of a smart contract in a blockchain decentralized network. Developers can redeploy a smart contract so as to eliminate outdated and deprecated storage and application logic from updated contracts; however, there is a dangerous point with regards to the security of smart contracts. This is called Reentrancy attack. The reentrancy attack can be eliminated with the concept of the deep role because a role that has been played before cannot be played by another core contract again. In the concept of deep roles concept (roles are playing roles), a core object can play roles, but played roles cannot play other roles. In this manner, smart contract security can prevent reverse calling for played roles by implementing role modeling.

The rest of the paper is structured as follows: We emphasize the research problem in Chapter 2 and describe our motivation with research questions in SubChapter 2.1 to conduct this research. Then, we provide background of role abstraction in smart contract design, design pattern concept in Solidity language, computational cost in stateful on-chain smart contracts in Chapter 3. Chapter 4 will be relevant to the current challenges that we have faced during the research study and the limitations of the study will be listed. After we have emphasized the key takeaways from related studies in Chapter 5, we will analyze the implementation of the proposed design pattern in on-chain smart contract programming. In Chapter 6, we will give details of implementation. In Chapter 7, we will discuss key findings regarding the conducted research with a list of conclusions to enlighten the reader. Finally, in Chapter 8, further key points in regard to development and research will be listed.

2. Research Problem

In this chapter, we will define our research questions and the major problems that have motivated us to complete this paper. Roles are abstraction layers of an object-oriented approach to solve multiple problems, which are:

- Extending key abstractions of roles should be represented as an aspect of suitability and cost-effectiveness in a smart contract language such as Solidity. Which smart contract language features are necessary to represent key abstractions of roles?

- Although managing dynamic roles is an important role-based language feature. Role Object Pattern (ROP) suffers from object schizophrenia and one can solve this problem with delegation and forwarding in object-oriented programming languages.

- Role-level constraints are necessary to maintain constraints among roles without a context. How can we implement constraints between roles?

- By implementing subclassing in a role modeling scenario, we can implement the Role Object pattern recursively. What are the drawbacks and benefits in terms of performance and gas cost for Solidity on-chain programming language?

In addition to the main problem interests, we would like to focus on the following research question:

Research Question 1 (RQ1): How to identify different role subtyping through interfaces in Solidity?

Research Question 2 (RQ2): How can type safety be ensured statically?

Research Question 3 (RQ3): What are the benefits and drawbacks of proxy pattern and interface selector according to role modeling in on-chain contract languages?

2.1. Motivation

The main motivation of this paper is to provide a preview of one of the important design patterns for role modeling, which is the Role Object Pattern (ROP). By implementing the extended version of this pattern, we will elaborate on possible implications while extending the design pattern with contexts in Solidity language. ROP focuses on the dynamicity of a role insertion into the system of the main design. Even if we have dynamic design patterns such as Proxy Delegate, Upgradable Standard for Proxy Delegate, and Eternal Storage, the ROP can support context which can provide a computational entity and grouping relationships in model-driven engineering.

3. Background

3.1. On-chain and Off-chain Smart Contract Programming Decentralized Networks

Concerning the programming concept in smart contracts, we have two different terms, which are on-chain and off-chain smart contract programming. The fundamental difference between off-chain and on-chain smart contract programming is being connected with an external service that is not part of a blockchain network.

A smart contract can check preconditions and postconditions in the on-chain network. On-chain smart contract programming has some advantages, which are:

Transactions are executed in an on-chain database, which is called blockchain network.

The state of the blockchain and smart contracts can be tracked by means of on-chain events.

With *event* and *emit* keywords, one can implement an advanced logging system without implementing third-party logging solutions. Beyond that, one can realize a domain-specific language for logging in the Ethereum

network that works with Solidity language to get information about transactions, transaction receipts, and states[2].

Off-chain data can be harmonized with external data sources such as specialized streaming data platforms, key-store databases, object databases, relational databases, interplanetary file systems or file regular file systems. The main difference between off-chain data and on-chain data is to manage data sources through a blockchain network or manually. Another property regarding off-chain data is supporting non-Turing complete smart contract languages. For instance, Bitcoin has an internal smart contract script language, however, it does support basic variable assignment without creating a loop and reference types.

3.2. Dynamic Contract Approach with Proxy Pattern is Solidity

Proxy patterns have been invented and implemented to realize hot bug fixes in Solidity programming since a contract code is immutable after deployment. Generally, the concept is widely used for gas savings and dynamic contract upgrading. After deploying a proxy contract, all messages will be transferred to the corresponding address, which can be a new version of a contract. In essence, we have three different types of proxy patterns[3].

- **Eternal Storage**: Updated contract remains in a blockchain network with old contract data layer and the data layer might consist of user information, account balances, or references to other contracts.
- **Unstructured Storage**: In this type of proxy contract, we need to follow the structure of reference data (attributes of a struct type) whether in the right order or not. Since the Solidity language sets up variables in a contract sequentially, the caller contract should move the references of storage variables (state variables) from the callee contract. The drawback of the pattern is that cannot be implemented to reference data structure of Solidity such as mapping and structs.
- **Inherited Storage**: This type of proxy contract ensures that the order and state of storage variables between caller and callee will be the same. The main aim is to protect the data layer of a smart contract without inserting complicated assembly code blocks in a smart contract[4].

All of the above-mentioned patterns require low-level dynamic calls that only should be invoked by experienced developers because they can easily contain a code snippet that has a hard-to-find bug. The main challenge in upgraded contracts is to preserve old storage (data layer) with updated contracts (application layer). All of the described patterns can be used to implement dynamic smart contracts; however, the main difference between them is being asset cost and handling storage of contracts because they share both proxy and contract behavior [5].

We have a couple of dynamic proxy patterns standards as below [6] :

Diamond pattern, Multi-Facet Proxy (EIP-2535): It solves the maximum contract size limit in the Ethereum world. A diamond pattern provides a way to organize smart contract code and smart contracts can be assigned as upgradable and immutable for the future. Moreover, the incremental upgradable smart contract is possible so that one can take the altered part of a smart contract is possible so that one can take the changed part of a smart contract, and can assign this part as upgradable by means of Diamond Pattern[7].

Transparent Proxy Pattern: The goal of the proxy pattern is to make indistinguishable an externally owned account with actual logic contract [8]. In order to prevent proxy selector clashing, which means that the same function signatures should be controlled in external contracts as well so that the transparent proxy pattern can be used because one can make a transaction without an admin of the proxy contract.

Universal Upgradable Proxy Standard (UUPS) (EIP-1822): This relies on a standard called EIP-1822[9]. In this standard, authors have two essential motivations which are [10]:

- Easy to deploy and maintain proxy and logic contracts.
- Standardization of proxy contract implementation by verifying the bytecode used by the Proxy Contract.

3.3. Role and Core Objects in Solidity

Although object-oriented programming solves major problems in software development, abstraction of objects and dynamicity have not been properly addressed and these are still hard to solve with object-oriented programming.

2 https://github.com/ChrisKlinkmueller/Ethereum-Logging-Framework
3 https://blog.openzeppelin.com/proxy-patterns/
4 http://blog.openzeppelin.com/upgradeability-using-unstructured-storage/
5 https://blog.openzeppelin.com/upgradeability-using-unstructured-storage/
6 https://blog.logrocket.com/using-uups-proxy-pattern-upgrade-smart-contracts/
7 https://eips.ethereum.org/EIPS/eip-2535
8 https://blog.openzeppelin.com/the-transparent-proxy-pattern
9 https://eips.ethereum.org/EIPS/eip-1822
10 https://eips.ethereum.org/EIPS/eip-1822

Single Role Type: All role features incorporate into one single role type. For instance, engineers, salesman, and directors can be differentiated by job descriptors and description IDs [3]. If each of the occupations has different features, different types can be generalized by means of interfaces from base interface classes. This concept is similar to role subtyping.

Role Subtyping: We can represent the many roles of an object in a smart contract language by making a subtype for each role [3].

Role Object: Common features can be inserted into a host object with a separate role object. In this case, occurrence constraints are hard to achieve, but it is easy to implement with client applications that work with host objects.

Role Relationship: Roles can be represented by many role objects. If a host object may have more than one role object, a role relationship comes into the game to represent occurrences between role objects.

4. Limitations and Challenges

This study is limited to the on-chain smart contract programming language that has object-oriented features to analyze and discuss the results of role modeling mapping in the object-oriented programming world. We have created qualitative and quantitative research parameters to test role-based applications. We exclude the networking and consensus layer of blockchain technologies because they are irrelevant to the implementation of ROP.

ContextROP is supporting only a static view of the context, which means that roles cannot be migrated from one context to another dynamically. We will mainly discuss on-chain crypto assets (gas cost optimization with role object pattern to reduce deployment, operational and computational costs of on-chain smart contracts. One of the biggest challenging points is to manage immutable smart contracts integrating role modeling. Roles can be assigned dynamically and static representation with interfaces does not provide all of the role-based modeling features such as dynamic loading, deep-role playing, and dynamic role type without interface definition.

One of the language limitations is the immutability of smart contracts in the runtime and advanced role-based type safety. Even if the Solidity programming language is statically typed, role subtyping could not be ensured with an extra effort of type safety structure. Another limitation can be mentioned regarding contexts, and one cannot easily describe constraints between roles. For instance, occurrence constraints can be implemented employing abstracted data types such as Set, HashSet, and HashMaps by checking the maximum or minimum number of occurrences at runtime. However, Solidity offers *Mappings* abstract data structure with limited functionality of *Map* data structure.

Another language limitation is the lack of *isInstance* or instance*Of* keyword like in an object-oriented language to provide type safety. Programming languages support static type safety for primitive data types such as integer, boolean, float, and double. If a developer wants to create an abstract type from a class, the possessiveness of a new object with an abstract type should be assured. Unlike Mappings and structs are major abstract non-intrusive types in Solidity language, one can create abstract new types with single, multiple, multi-level, and hierarchical inheritance systems with this language.

The last limitation is the dynamic deployment of a smart contract. A single smart contract has the limitation of a maximum 24KB contract size, which means that smart contracts cannot have a complex role-based application with extensive computational loops. ContextROP design pattern does not address the smart contract size limitation.

5. Related Work

The main paper about the role object pattern has been proposed by Bäumer et. al. [4] and they have claimed that attached role objects represent a role that can be played by an object in a client's context. In their design pattern description, they have described role objects and core objects.

Stolz and Steimann have proposed lightweight role objects that can refactory roles implemented as subclasses of a role player letting instances of three role objects share state and identity with an instance of the role player object [11] . Moreover, the authors have used the method that is called Replace Inheritance with Delegation Refactoring (RIWD) to allow reuse without role subtyping in order to avoid cumbersome and bloat interfaces that define role subtyping and supertyping [12] .

Martin Fowler has divided up different parts of role-based modeling, which are Single Role Type, Separate Role **Type, Role Subtype, Role Object, Role Relationship, Internal Flag, Hidden Delegate, State Object, Explicit Type Method, and Parameterized Type Method** [3]. The main motivation of this paper is to distinguish role-based modeling features from different conceptual ideas by emphasizing the main takeaways. In the *Conclusion* chapter of the study, the author stated that every selection of a design pattern in role-modeling has a trade-off in software modeling and programming. Another fact from the paper can be deducted as role-playing assumptions that may not be true for all condi-

11 https://www.fernuni-hagen.de/ps/prjs/IROP/

12 http://www.feu.de/ps/prjs/RIWD/

tional testing. A role player can be important for an individual use case, but another user may not need to constrain the role player because the role objects itself is more important than the role players in this case.

Stephan Hermann has claimed that roles define the intersection of objects and contexts. Contexts can be grouped by static and dynamic views, which means that either a set of roles can be assigned to a static context or they can migrate from one context to another [6]. In this paper, the author proposed a language called ObjecTeams that has the capability to group a number of roles into a context, more precisely in the paper is called team [6].

Steimann and Urs Stolz [7] have proposed refactored role object pattern by way of intensive usage of subclassing in an object-oriented language. These subclassing methods can be listed as follows [7]:

- **Entity Type:** A base class can be renamed as an entity type in order to create abstracted role types from a core component class.
- **Base Interface:** From an interface, an entity type and abstract role type can be created to constitute concrete role types.
- **Component Type:** This is the regular way of creating role types in the Role Object Pattern. It inserts a new abstract class between the entity class and intermediate subclasses to create role classes that do not change the behavior of a program because it does not add anything [7]

In this paper, the main idea is to replace inheritance with delegation refactoring to add roles to the component core. Cabot and Raventos emphasized the importance of the *Role as Entity Types* pattern that can be useful to represent roles while a role-based application requires full expressiveness [5]. Cabot and Raventos have started to list role features such as ownership, control, role-playing, role identity, adoption, and relationship. In the design and implementation phase, they have been categorized into three major topics, which are [5]:

- **Roles as Subtypes Pattern:** Roles can be designed by subtypes of a base class. For instance, Teacher and Student can appear as subtypes of Person class.
- **Roles as Interfaces Pattern:** Roles are represented as interfaces and this study utilizes the approach in the implementation. We can specify entity types that play a certain role through interfaces.
- **Roles as Reified Entity Types Pattern:** Roles are represented as reified entity types with a relation type. A Student type can represent a relationship between University and Person even if it is not clear possessive of role type to University or Person.
- **Roles as Participant Names Pattern:** A role is barely represented as a name assigned to an entity type in a relationship type. For instance, Project Manager and Branch Manager cannot be occurred in the same conceptual schema since in this case, a role cannot play other roles.

Steimann [6] claims that interfaces are a prominent Object-Oriented programming concept since they allow decoupling of implementation [8]. One can declare every variable and parameter with an abstract type in order to realize roles as interfaces. In the definition UML metamodel of the paper, a merger module can merge *Interface* and *ClassifierRole* to a new metaclass called *Role*. In the conclusion of the study, the conceptual representation of roles with interfaces does not cover all features of the role concept.

Wöhrer and Zdun [9] summarize a set of patterns such as contract register pattern, contract relay pattern, and satellite pattern. Through the contract register pattern, contract participants can be pointed to the latest contract version. The register contract keeps track of different versions (addresses) of a contract. Moreover, a contract relay pattern can be useful to handle the update process of a contract [9]. By means of the satellite pattern, one can store addresses of them in a base contract that allows for modification and replacement contract functionality.

6. Implementation

Implementation is twofold for the application of Solidity programming. Dynamic proxied and static interface separator for team activation. We would like to list both features to evaluate differences in asset cost (gas cost), and performance evaluation in deployment. The static interface has been implemented with a standard interface detector that is called EIP-165.

We have major 5 different classes, which are:

- ERC165 (EIP-165)
- Component
- ComponentCore
- ComponentRole
- Team
- ExtendedRole
- Each of them has a different purpose while creating role-based applications in specified contexts, which are:

ERC165 (EIP-165): Interfaces should be identified and differentiated in Solidity programming. The main aim is to detect if a contract implements any given interface[13].

13https://eips.ethereum.org/EIPS/eip-165

In the ERC165 contract, there is a function called *supportsInterface()* that takes an *interfaceID* bytes32 format.

Component: This is an interface that represents a key abstraction for *ComponentRole* and *ComponentCore* to define adding and removing role objects. *Component* is the base entity that provides a role management interface.

ComponentCore: A core object creates a role object from this «abstract» contract to play a role and it implements role management protocol through the *Component* interface.

ComponentRole: ComponentRole is the main component that can create role objects from core objects.

Team: Team smart contract provides context-based grouping for role-based applications. Principally, the Team represents the context concept that can activate and deactivate to single or multiple roles accordingly.

Extended Role: This smart contract utilizes the approach of a dynamic proxy pattern that can help us to deploy an updated contract with a low gas cost. ExtendRole can be used in the UUPS Proxy Contract implementation in order for providing dynamic upgradable contracts.

```
1  Interface Component {
2          function addRole(bytes32 spec, address
           role) external;
3          function removeRole(bytes32 spec) ex-
           ternal;
4          function isPlayingRole(bytes32 spec)
           external;
5          function getRole(bytes32 spec) external
           returns (address);
6          function activateTeam(address team)
           external;
7          function deactivateTeam() external;
8          function getActivateTeam() external
           view returns(address);
9  }
```

Listing 1: Component Interface in ContextROP

As listed in Listing 1, we have major functions activating context and dispatching role objects to them with bytes32 spec addresses. Bytes32 dynamic array of bytes can be selected because the data type can be utilized in function arguments to pass arguments and return a result from a contract.

```
1  library InterfaceCodes {
2          bytes4 constant COMPONENT_ID = type(Compo-
           nent).interfaceId;
3          bytes4 constant COMPONENT_ROLE_ID = type(Com-
           ponentRole).interfaceId;
4          bytes4 constant TEAM_ID = type(TEAM).interfaceId;
5  }
```

Listing 2: InterfaceCodes for EIP165 Contract Separator

As for Listing 2, *InterfaceCodes* can be customized to distinguish abstract types from base classes from each other. Interfaces are identified as a set of function selec

tors in the Application Binary Interface (ABI) definition of Solidity programming language. To prevent invoking different function signatures as if they were the same, *bytes4* of the function signature hash should be used with customized *InterfaceCodes.*

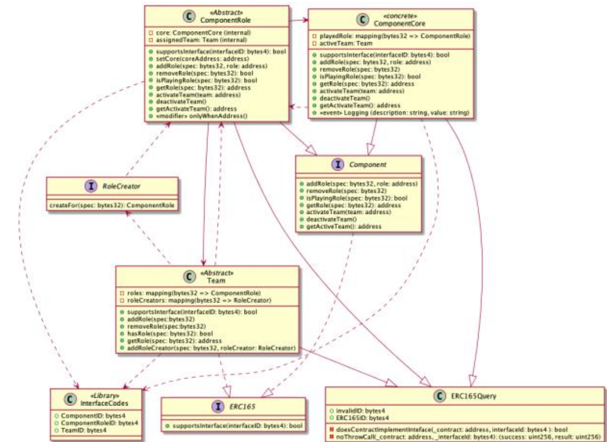

Figure 1: UML Class Diagram for ContextROP Standard Interface Detector (EIP-165)

As shown in Figure 1, we have different modules to create the fundamental requirements of the ContextROP design pattern that consists of ERC165, InterfaceCodes, Team, RoleCreator, ComponentRole, ComponentCore. Team, ComponentRole, and ComponentCore should inherit a set of functions such as doesContractImplementInterface, and noThrowCall. Chiefly, these two functions can control interface identification numbers to simulate functions like typeOf or isInstance in object-oriented languages. Since natural type safety mechanism is not found in on-chain smart contract language, namely Solidity, developers can provide the subclassed type safety by way of interface detectors.

To connect through an externally owned account (EOA) to the Context-ROP application, load, and deploy methods should be given. These methods are overloaded methods with function signatures that take contractAddress, Web3j credentials, unit value of the gas price, and gas limit. After one loaded a contract, they might be deployed with the same aforementioned parameters through RemoteCall.

Order to reach on-chain ContextROP application by means of a general-purpose language. Contract address, credentials, and contract gas provider should be defined by externally owned accounts.

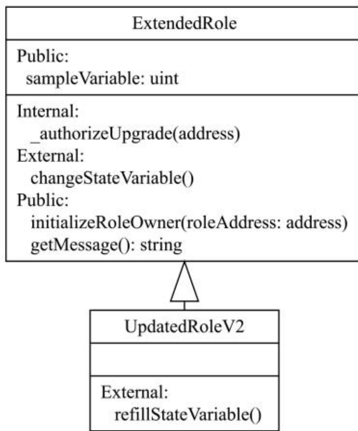

Figure 2: UUPS Diagram for Roles

As depicted in Figure 2, one of the concepts is the UUPS Proxy Pattern that originated from EIP-1822. This pattern relies on the storage holder (proxy) and logic contract implementation. In the proxy contract, the contract stores storage variables that can be used by a logic of an external contract. A state variable or storage layout organizational pattern is needed because Solidity's built-in storage layout system does not support proxy contracts [14]. Through advanced libraries, using *DelegateCall*[15] is not a hurdle that developers should cope with.

Figure 3: Sequence Diagram for Roles

As shown in Figure 3, we have different stages to interact with a role-based application from creation to termination. When a role creation function is invoked by an Externally Owned Account (EOA), a particular address should be given to the reciprocal function. It works with external clients or wallet accounts in the blockchain network. A smart contract naturally cannot activate another on-chain smart contract in a blockchain network. After adding a particular role, the role can be played with its name in a registered team (context) via *activateTeam(roleName)* function. A strict condition in the sequence diagram is to deny role-playing after deactivating with the function called *deactivateTeam().* Analogous to creating roles, accessing roles can invoke similar methods with *bytes4* or *bytes32* variables for Ethereum VM addresses in the shared memory.

7. Evaluation

In this chapter, we would like to evaluate the ContextROP application that we have created in Solidity Programming Language. In role-based definitions of Steimann [10], Kühn (PhD Thesis, A Family of Role-based Languages, Thomas Kühn), we can list 26 different statements. These statements cover most of the role modeling features from two different research studies. Result of the qualitative evaluations can be seen in Figure 4 and Listing 3.

Role features	ROP (1999)	PowerJava (2006)	Rumer (2007)	NextEJ (2009)	SCROLL (2017)	OT/J (2005)	Chameleon (2003)	JavaStage (2012)	ContextROP (2022)
1.	■	■	■	■	□	■	■	■	■
2.	■	⊞	□	⊞	⊞	⊞	□	□	■
3.	■	■	■	■	■	■	■	■	■
4.	■	■	■	■	■	■	■	□	■
5.	■	■	■	■	■	■	■	□	■
6.	⊞	□	□	□	⊞	□	□	■	⊞
7.	□	■	■	■	□	■	■	■	□
8.	□	■	■	■	□	□	□	■	□
9.	■	■	■	■	■	■	■	□	■
10.	■	■	■	■	■	■	■	■	■
11.	■	■	■	■	■	■	■	■	■
12.	⊞	■	⊞	■	⊞	■	■	■	⊞
13.	■	□	■	■	■	□	□	■	■
14.	■	⊞	■	⊞	■	⊞	⊞	□	■
15.	□	■	■	■	□	■	■	■	□
16.	□	□	□	□	□	□	□	□	□
17.	□	□	□	□	□	□	□	□	□
18.	□	⊞	⊞	⊞	⊞	■	□	□	⊞
19.	□	■	□	■	■	■	□	□	■
20.	□	■	■	■	■	■	□	□	■
21.	□	■	⊞	■	⊞	□	□	□	⊞
22.	□	□	■	□	□	■	□	□	□
23.	□	□	■	□	□	■	□	□	□
24.	□	■	■	■	⊞	■	□	□	⊞
25.	□	□	■	□	■	■	□	□	■
26.	□	■	■	■	■	■	□	□	■

Figure 4: Assessment of approaches with regards to developing roles at runtime using 26 classifying features taken from [10] [11]. Features are completely supported (■), partially supported (⊞), and not supported (□) features.

14https://eips.ethereum.org/EIPS/eip-2535

15https://solidity-by-example.org/delegatecall/

A detailed list of the role features from [10] [11]:

1. A role comes with its own properties and behavior:
2. Roles depend on relationships:
3. An object may play different roles simultaneously
4. An object may play the same role several times, simultaneously
5. An object may acquire and abandon roles dynamically.
6. The sequence in which roles may be acquired and relinquished can be subject to restrictions.
7. Objects of unrelated types can play the same role.
8. Roles can play roles
9. A role can be transferred from one object to another.
10. The state of an object can be role-specific.
11. Features of an object can be role-specific
12. Roles restrict access.
13. Different roles may share structure and behavior.
14. An object and its roles share an identity.
15. An object and its roles have different identities.
16. Relationships between roles can be constrained.
17. There may be constraints between relationships.
18. Roles can be grouped and constrained together.
19. Roles depend on compartments.
20. Compartments have properties and behaviors like objects.
21. A role can be part of several compartments.
22. Compartments may play roles like objects.
23. Compartments may play roles that are part of themselves.
24. Compartments can contain other compartments.
25. Different compartments may share structure and behavior.
26. Compartments have their own identity.

Listing 3: Quantitative Evaluation for the Context-based Role Object Pattern (ContextROP)

In Table 1, we see deployment and execution costs to understand the difference between methods in ContextROP implementation. The UUPS and normal methods have already been discussed in previous chapters. In this section, we had a use case that simulates a banking application with a set of investors and borrowers. Borrowers can borrow a certain amount of money from Banking Institution and discharge the debt before the overdue payment. Borrowers and Investors have a creator method that will associate the ContextROP smart contract package to create abstract role types. In the following Table 1, readers can see the deployment and execution costs during the interaction between roles and objects.

The results have been taken from Remix (v0.25.1) with Hardhat Provider by interacting same functions with different methods. The calculation unit has been given as Gas. Deployment cost shows us the cost of contract deployment into the network. Transaction cost refers to the cost of method interaction with a parameter in on

chain environment. Normally, the transaction cost is utilized for sending the contract code to the blockchain, but we did not use that meaning. Execution cost is based on the cost of computational operations.

One of the important results is the deployment cost of EIP-165 for both roles is lower than UUPS Pattern roles. This can be understood because we are using additional libraries to implement UUPS Pattern; however, incresing gas costs can make the development process expensive. Even the execution cost has been doubled while implementing UUPS Proxy since it can solve the contract maximum size problem. Deployment and Execution costs are the initial cost to interact the contract with the blockchain network. Transaction cost should be considered when producing the cost to invoke a specific method. In Table 1, transaction costs of different role methods are similar to each other and one can say that runtime execution cost is not different from each other.

Method	Deployment Cost	Execution Cost	Transaction Cost
EIP-165 (Role Investor)	1333374 gas	1159455 gas	Invest () method - 54123 gas
EIP-165 (Role Borrower)	1467419 gas	1276016 gas	Borrow() method - 54145 gas
UUPS Pattern (Role Investor)	2863718 gas	2490189 gas	Invest() method – 54174 gas
UUPS Pattern (Role Borrower)	2987612 gas	2597923 gas	Borrow() method - 54106 gas

Table 1: Assessment of the gas cost while executing different methods in ContextROP Standard Implementation.

8. Discussions and Conclusion

Contexts and Roles are the modeling nature of programming languages and they can be used for producing key abstractions to model abstract states and behavior. While implementing this pattern, the most prominent feature of the on-chain smart contract programming occurs in non-intrusive dynamic contract behavior without coping with low-level calls such as D*elegateCall*. Even if we realize *DelegateCall* by way of proxy patterns such as diamond standard pattern, unstructured storage pattern, or transparent proxy pattern.

The first finding is that the type-safety can be implemented with Interface Separator (EIP-165) in role-based applications with Solidity programming language. Implementation of proxy pattern with the EIP-165 concept can reduce a great deal of gas cost initial development stage.

Moreover, it can be more flexible to deploy a contract in case of bug fixing and regular maintenance. However, hardcoded *InterfaceCodes* can cause the function selector clashing. To prevent this, a random UUID (Universal Unique Identifier) can be generated by external libraries.

The second finding is that deployment and execution costs can be reduced through the usage of the EIP-165 standard with a role-based application. Even though there is no difference between the transaction cost of similar function signatures, EIP-165 originated smart contracts can reduce deployment and execution costs more than pro contracts do. However, proxy patterns can solve the contract size limitation, unlike EIP-165 contracts.

The third finding is that most of the role features (14 features) can be completely supported by Solidity programming languages because it is utilizing object-oriented programming techniques. Abstraction of core objects and role subclassing are the fundamental techniques for role-based programming and most smart contract programming languages cannot provide OOP concept except for EOA client source code itself (not in on-chain contract language).

The fourth finding is that trust enabling, and auditable actions are a strong necessity for role-based applications because participants of external clients (roles) should be trustable and auditable. Smart contracts can provide trustable and auditable context-awareness by *Modifier* keyword (function modifiers) *to* role objects by enabling a trust layer on the network.

Additionally, source code of ContextROP can be found at the following link: https://github.com/zointblackbriar/Smart_Contract_Examples/tree/development/Context-Based-CROP-Solidity.

9. Future Work

Different adaptations, test cases, and a set of use cases have already been implemented; however, some parts are still open to research on them. Dynamic contexts can be added to migrate roles from one context to another. Even though there is no consensus on how to create a role, role features from various research studies can be implemented in an object-oriented approach with on-chain smart contracts.

Refactoring roles as subclasses for entity types and hierarchical interface methods can also define role object pattern with context. Different patterns can be compared to aspects of execution and deployment cost in the blockchain network.

Acknowledgment

The author would like to thank his supervisor, Prof. Dr. Uwe Aßmann, for the patient guidance, encouragement, and comments he provided to shape his paper vision. This work is funded by the German Research (DFG) within the Research Training Group Role-Based Software Infrastructures for Continuous Context-Sensitive Systems (GRK 1907, TU Dresden, Software Technology Group, Nöthnitzer Straße 46, 01187, Dresden).

References

[1] Herrmann, S. (2010). Demystifying object schizophrenia.

[2] 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: 2901-2925.

[3] Fowler, M. (1997). Dealing with roles, 97.

[4] Bumer, D.; Riehle, D.; Siberski, W. and Wulf, M. (1997). The Role Object Pattern.

[5] Cabot, J. and Raventós, R. (2006). Conceptual Modelling Patterns for Roles, Journal on Data Semantics - JODS 3870: 158-184.

[6] Herrmann, S. (2007). Programming with Roles in ObjectTeams/Java, Applied Ontology 2 : 181-207.

[7] Steimann, F. and Stolz, F. U. (2011). Refactoring to Role Objects: 441–450.

[8] Steimann, F. and Wissensverarbeitung, R. (2000). Role = Interface - A merger of concepts.

[9] Wohrer, M. and Zdun, U. (2018). Design Patterns for Smart Contracts in the Ethereum Ecosystem: 1513-1520.

[10] Steimann, F. (2000). On the Representation of Roles in Object-Oriented and Conceptual Modelling, Data & Knowledge Engineering 35: 83-106.

[11] Kühn, T.; Leuthäuser, M.; Götz, S.; Seidl, C. and Aßmann, U. (2014). A metamodel family for role-based modeling and programming languages: 141-160.