Aspect Programming: A programming model enabling native extensions on the blockchain

Honglin Qiu, Jack Li, Mike Ma, Kevin Yang, Jerry Li

July 14, 2023

Abstract

We present Aspect Programming, a programming model for Artela Blockchain[1] that enables native extensions on the blockchain. Aspect is the programmable extension used to dynamically integrate custom functionality into the blockchain at runtime, working in conjunction with smart contracts to enhance on-chain functionality. The distinguishing feature of Aspect is the ability to access the system-level APIs of the base layer[1] and perform designated actions at join points throughout the transaction's life cycle. Smart contracts can bind specified Aspects to activate additional functionality. When a transaction invokes these smart contracts, it interacts with the associated Aspects. Aspect Programming is also designed as a universal programming model applicable to any other blockchains[2]. With Aspect Programming, developers can implement basic logic in smart contracts and extend additional features in Aspects to build feature-rich dApps beyond the capabilities of EVM.

1 Introduction

The smart contract has become a predominant method for developing decentralized applications (dApps), and the Ethereum Virtual Machine (EVM) is widely integrated into various blockchains. Despite notable advancements in scalability within the blockchain realm, smart contracts, especially EVM-based ones, are still limited in terms of fully realizing dApp functionality due to the restricted extensibility of the underlying layer. Aspect Programming aims to address this limitation, enhancing the programmability of the blockchain while ensuring compatibility with existing smart contracts and their ecosystems.

In this paper, we present Aspect Programming, a programming model that enables native extensions on the blockchain. Our discussion of Aspect Programming includes the following sections:

1. Offer a brief overview of the issues of blockchain extensibility and present the native extensions as a solution. We explain how native extensions address these issues and discuss their advantages for both developers and users. (Section 2)

2. Provide an overview of how to implement native extensions with Aspect Programming and introduce the structure and key features of Aspect Programming. (Section 3)

3. A deep dive into the technical details of Aspect Programming, covering its in-depth model design (Section 4) and implementation (Section 5), supplementing with miscellanea about design thinking (Section 8). It aims to foster a clearer understanding of Aspect Programming and its functionality.

4. Introduce what innovative application features Aspect brings to dApps, and describe the impact on the broader blockchain ecosystem, including infrastructure, middleware, and application building. (Section 6)

[1] A core basic layer with system-level modules. (mempool, consensus, networking, etc)
[2] Including layer1 blockchain and layer2, which provide smart contract functionality

5. Conclude by summarizing what we have done for Aspect Programming, describing the plans for its ongoing enhancement and adoption, and outlining its possible future features. (Section 7)

2 Extensibility of Blockchain

In this section, we describe the extensibility issues of blockchain and introduce a new way to address them.

Extensibility[1] is a measure of the ability to extend a system and the level of effort required to implement the extension for customized functionality. Blockchain extensibility [**?**] refers to the ability to support user-defined functionality either through a built-in blockchain implementation or through extension mechanisms that allow developers to craft applications beyond the original intent of the platform.

Advancement is underway in the realm of blockchain extensibility. Bitcoin [**?**] implements a built-in distributed ledger that enables basic functionality with Bitcoin script. It marks the initial steps toward blockchain extensibility. Later, Ethereum [**?**] implements a built-in distributed state machine (EVM) that enables enriched functionality with stateful smart contracts. It marks a big step forward in blockchain extensibility for adding user-defined functionality at runtime. Since then, EVM-based smart contract blockchains have proliferated.

However, the extensibility of EVM-based smart contract blockchains is still limited. In this section, we discuss the extensibility problems of smart contract blockchains, the ways to address them, and Artela's solution. Artela aims to maximize the extensibility of smart contract blockchains.

2.1 The Extensibility Problems of Smart Contract Blockchain

The Ethereum Virtual Machine (EVM) is a widely adopted execution environment for smart contracts, used across numerous blockchains. While the EVM is designed to be Turing-complete, blockchains relying on the EVM face challenges in supporting complex dApps with advanced functionality. There are three main limitations related to extensibility:

1. **Limited functionality of smart contracts:** EVM programs are restricted from using loops and must terminate within a specified number of steps. As a result, translating intricate programs that include a large number of instructions and asynchronous tasks from one Turing-complete language into an equivalent EVM program can be a challenge. Developers have to minimize computation steps to avoid exceeding gas limits.

2. **Limited extensibility of EVM:** The precompiled smart contract is prohibited from being built at the user level, and its ownership is strictly controlled by the developers of the platform. The precompiled smart contract is enabled by EVM for on-chain program functionality, but it needs permission from the blockchain. EVM extensibility is limited for smart contract developers.

3. **Limited customization outside EVM:** The process of transactions outside EVM is statically pre-configured and cannot be customized. In addition to the state transition within EVM, outside the EVM, transactions undergo other processes including pre-execution validation, queuing, post-execution data indexing, and so on. These processes are pre-set and cannot be modified permissionlessly on generalized blockchains.

If a dApp requires the implementation of advanced features on-chain but is restricted by the aforementioned issues, the developer may need to either construct an independent, application-specific blockchain or propose an Ethereum Improvement Proposal (EIP) and await its implementation, which is a lengthy process.

2.2 The Ways to Address Extensibility Problems

The key to addressing these problems lies in creating a robust execution environment that not only accelerates performance, facilitating the crafting of complex logic in smart contracts, but also supports customization of various other transaction lifecycle processes. Our research identifies two fundamental methods to achieve this goal: the Superior-VM approach and the Extension-based approach.

2.2.1 Superior-VM Approach: Developing an Enhanced Smart Contract VM

The concept of creating a new, faster virtual machine (VM) capable of implementing additional transaction processes is a widely accepted solution. Move, a new programming language, is used to implement custom transaction logic and smart contracts within the Libra protocol. Substrate's runtime enables users to define entire transaction processes, not just state transition functions.

EVM, a full-fledged VM, encompasses a rich ecosystem comprising numerous developers, extensive tooling, middleware, and widely adopted dApps. Several major public chain networks, including Layer2, integrate EVM to provide a mature development experience for their users. In the future, the development of enhanced smart contract VMs may become essential, but this journey toward VM maturity will be a long and evolving process.

2.2.2 Extension-based Approach: Providing Support for Extensions alongside the Smart Contract VM

With this approach, one can write a specific type of program, or an 'extension,' which performs faster and enables more states than EVM smart contracts to execute complex tasks that EVM struggles with. For example, in Avalanche, stateful precompiled contracts are used to customize the execution environment for Subnet and can be invoked by its EVM smart contracts. These precompiled contracts operate at native speed and permit the use of additional libraries.

By utilizing the extension-based approach, blockchains remain compatible with EVM or other existing smart contract VMs while enabling customization. It means that developers can build dApps with more advanced features on top of the existing smart contract ecosystem.

2.3 Our Solution: Native Extension and Aspect Programming

In the extension-based approach, the precompiled extension mode is currently widely adopted. However, the precompiled solution doesn't support adding user-defined functionality at runtime, and it doesn't support customization outside EVM. As a result of these limitations, the precompiled solution is primarily employed at the system level to address specific issues rather than at the user level to tackle a wide range of problems (to support more possibilities of innovation).

To address those limitations, we propose the implementation of programmable native extensions on the blockchain, which enables developers to develop, deploy, and utilize user-level extensions with the features of being more blockchain-native, dynamic, permissionless, local-scope, and generic.

- **Blockchain-native:** Native extensions are programs that run fully on-chain, executed deterministically by all validators in the network.
- **Dynamic:** Different from precompiled contracts, native extensions are dynamic programs that can be deployed onto the blockchain at runtime.
- **Permissionless:** Users can define and deploy their own programs. The blockchain acts solely as the execution environment, without any restrictions or permissions imposed by a centralized authority.
- **Local-scope:** Smart contracts locally activate native extensions, ensuring that the effects of native extensions are within the local scope in which they are activated, not the global scope.
- **Generic:** Native extensions provide access to the entire blockchain state and runtime context,

allowing developers to customize outside EVM and participate in controlling the transaction lifecycle to realize generic and versatile customization.

With the native extension, users can independently introduce system-level functionality to their dApp. These native extensions seamlessly collaborate with smart contracts, enhancing the overall functionality and potential of the dApp ecosystem. For example, users can develop a runtime protection native extension for their DeFi smart contract to counteract hacking attempts. This extension can monitor unexpected fund flows from the vault by accessing the entire call stack and tracking state changes during the transaction. Furthermore, users can deploy a native extension that incorporates a new curve algorithm to enhance their verification process, overcoming limitations imposed by fixed algorithms used by nodes.

To implement native extensions on the blockchain, we present Aspect Programming, a programming model that facilitates the creation of native extensions on the blockchain. In the next section of this paper, we will provide a detailed introduction to Aspect Programming.

3 Aspect Programming Overview

The core principle of the Aspect Programming framework is the Join Point Model (JPM). This model defines three key components:

JoinPoint: It specifies where the Aspect can run. A join point represents a specific point in the transaction processing flow and the block processing flow. It acts as a hook, allowing additional functional logic to be added at these points.

Aspect: It specifies the code to run on the join points. An Aspect can access the runtime context and make system calls, enabling it to participate in transaction lifecycle management.

Binding: It specifies when the Aspect can run. Smart contract owners have the freedom to bind Aspects to specific join points with their smart contracts. When transaction processing steps reach these join points, the bound Aspects are triggered.

To provide a clearer understanding of how Aspect Programming works, let's walk through the transaction processes using an example. In this scenario, there is a smart contract with a vault function and a large deposit. The goal is to protect this smart contract at runtime to avoid illicit attacks that might illegally move out deposits. For this purpose, an Aspect is designed to be responsible for monitoring and verifying any deposit changes in the vault after smart contract execution. If there is an unexpected fund flow, the Aspect will revert the suspicious transaction. The Aspect execution is triggered after the smart contract execution when a transaction calls the smart contract.

3.1 Aspect and Join Points

An Aspect is implemented as a class that extends the Aspect base class. It contains methods that represent join points where additional logic can be injected. Here's an example implementation of an Aspect:

```
class SecurityAspect extends Aspect {
  @joinpoint
  postTxExecute(jpCtx: JPContext) {
    if (jpCtx.currentCall().methodName() == "withdraw") {
      let mintAmount = jpCtx.currentCall().params()[1];
      let actualVaultEtherFlow = jpCtx.stateChange(jpCtx.tx.to()).ether().diff();

      if (mintAmount != actualVaultEtherFlow) {
        jpCtx.txControl().revert();
      }
    }
```

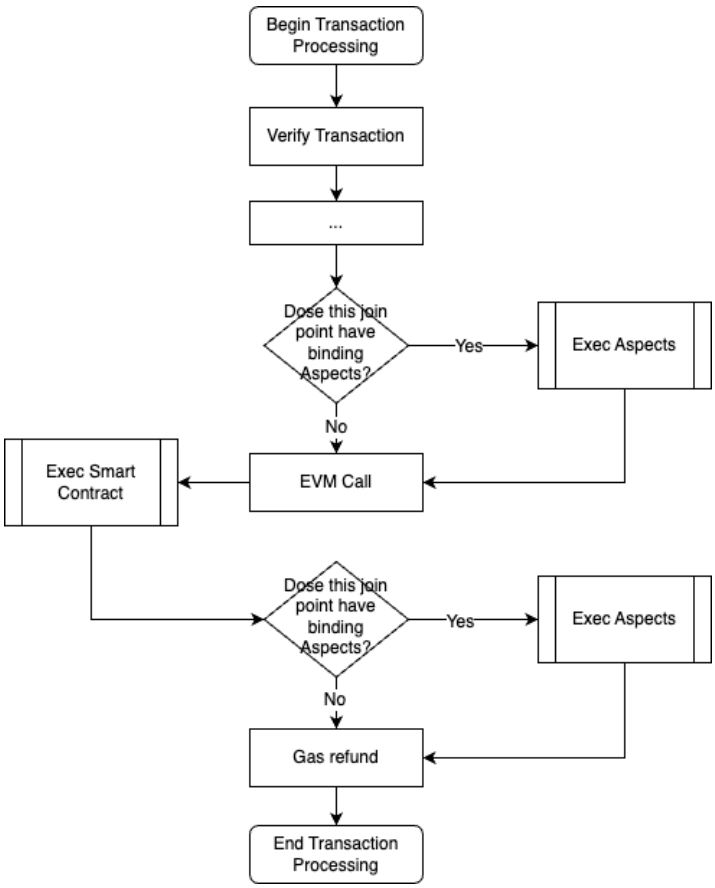

Figure 1: Join Point Model

```
  }
}
```

In this example, the `postTxExecute` method serves as an entry point for the join point, which is triggered once the Ethereum Virtual Machine (EVM) completes transaction execution. The method is decorated with the `@joinpoint` annotation to indicate that it should be registered as a join point.

The `JPContext` object represents the runtime context and provides information about the original transaction, transaction runtime details at the current process point, and APIs for controlling transaction behavior. It is passed as a parameter to the join point method, enabling interaction with the transaction.

The example Aspect checks if the `withdraw` function of the smart contract is invoked during transaction execution. If so, it retrieves the expected minting amount from the transaction parameters and the actual funds flow of the vault from the runtime context. If there is a discrepancy between the expected and actual amounts, indicating a potential coding error or a cyberattack, the Aspect triggers the `revert()` function provided by the transaction management object in the runtime context, resulting in the reversal of the transaction.

3.2 Deployment and Binding

To deploy an Aspect onto the blockchain, its bytecode is included in a deployment transaction. This transaction calls an Aspect system contract, which stores the Aspect's bytecode within the blockchain's world state. Validators within the blockchain gain access to this bytecode, allowing them to execute the Aspect's logic when triggered.

However, an Aspect is only executed when it is bound to a specific smart contract. The binding process involves the smart contract owner signing a binding transaction using their externally owned account (EOA), the same account used for deploying the smart contract. This binding transaction includes the smart contract address and the Aspect ID. When this binding transaction is executed, it triggers the Aspect system contract, establishing the binding relationship between the smart contract and the Aspect within the blockchain's world state. Only the smart contract owner has the authority to bind Aspects to their smart contract, preventing unauthorized binding attempts by other EOAs.

Once the deployment and binding transactions are successfully processed, the Aspect is deployed onto the blockchain and bound to the smart contract. This integration enhances the capabilities and security of the smart contract by incorporating the Aspect's functionality.

3.3 Execution with Aspect

When a transaction is initiated on a smart contract, the Aspect Programming framework runtime is activated. As the transaction begins execution, the Aspect runtime evaluates each join point to determine if there are any bound Aspects associated with the invoked smart contract. If an Aspect is bound to a join point, it is triggered upon the completion of the transaction execution.

To execute an Aspect, the Aspect runtime retrieves its bytecode from the blockchain's world state and loads it into a WebAssembly (Wasm) runtime environment. A context object is then constructed, providing relevant information about the transaction and the execution environment. Finally, the Aspect runtime invokes the entry function of the Aspect, triggering the execution of the Aspect's logic.

For example, suppose the smart contract's `withdraw` function contains logic that ensures the transferred funds always exceed the amount specified in the function parameter. If an Aspect is bound to the `postTxExecute` join point, it will be triggered at the completion of the transaction execution. The Aspect runtime retrieves the Aspect's bytecode, loads it into the Wasm runtime, constructs a context object, and invokes the Aspect's entry function. If the Aspect identifies a discrepancy between the expected and actual funds transfer, it can set a revert flag via the join point context. Upon detecting this flag, the Aspect runtime reverses the transaction, labels the result as a failure, and provides a rationale specifying that the transaction was overturned by the bound Aspect.

3.4 Conclusion

The Aspect Programming framework's core component is the Join Point Model (JPM), which provides hooks for injecting additional logic at specific points in the transaction and block processing flows. Aspects, which contain the additional logic, can access the runtime context and make system calls to participate in transaction lifecycle management. By binding Aspects to specific join points, smart contract owners can enhance the capabilities and security of their contracts.

In the following sections, we will delve into a detailed design of the Join Point Model and its implementation.

References

[1] Author Name. Title of the article. *Journal Name*, 1(1):1–10, 2023. 2