PREDA
A Distributed Programming Model for General Smart Contracts on Sharded Blockchains and Cross-Chain Bridges

PREDA DEV TEAM

PREDA, **P**arallel **R**elay-and-**E**xecution **D**istributed **A**rchitecture, is a novel programming model for general smart contracts running on multi-chain blockchain systems, in which transaction executions and ledger states are divided and distributed across chains. PREDA model decouples the schemes of such dividing/distributing with the architecture of the underlying blockchain system and the actual consensus protocols employed.

PREDA model divides the entire ledger to a number of non-overlapped *scopes*, which can be distributed and parallelized across chains. *Programmable scope schemes* are introduced to describe such distributed dividing, and to define ledger states and functions within each scope. Every scope is an independent sequential state machine that can be distributed and driven by an arbitrary chain in the system. All scopes are inherently parallelized by being distributed in multiple chains that operate in parallel.

Contract function invocations across scopes are allowed, which facilitate the interactions and coordinations of scopes. *Functional relay semantics* provide a systemic and legible way to express customized workflow that executes across scopes without involving underlying details of consensus systems and relaying mechanisms. Cross-scope invocations are asynchronous and are by design decoupled with the sharding structure, or bridging relationship, of the underlying multi-chain system.

1 INTRODUCTION

Smart contracts [3] provide an efficient and flexible way to define applications on blockchain systems, a.k.a. DApps. Listing 1 shows an example of a simplified contract for payment (ERC20) written in Solidity [8], which is the most widely adopted smart contract language. The code snippet defines a contract state, i.e. `balances` representing the balances of each corresponding address, and a contract function `transfer`, which is to transfer a given number of `amount` tokens from the transaction sender (`msg.sender`) to a specific recipient (`receiver`). A payment transaction is a piece of digitally signed data indicates an invocation of the function `transfer` with serialized function augments `(receiver, amount)`.

```
contract ERC20Basic is IERC20
{
    mapping(address => uint256) balances;
    function transfer(address receiver,uint256 amount) ...
    {
        require(amount <= balances[msg.sender]);
        balances[msg.sender] = balances[msg.sender] - amount;
        balances[receiver] = balances[receiver] + amount;
        return true;
    }
}
```

Listing 1. The code snippet of `transfer` function in an ERC20 contract in Solidity

In this example, the ledger state is a map from user (`address`) to their `balance` and, in any function, the entire states are available for reading and writing with a direct invocation that returns immediately. A smart contract is defined equivalently to a sequential state machine, which implies a simple but constrained programming model.

- **Sequential Execution** Each contract function invocation or every transaction must be executed sequentially to avoid concurrent access of ledger states, which is potentially unsafe.
- **Single-Box States** Since any function has direct and immediate access to any part of the states, the entire ledger states must be available in all nodes and kept synchronized as chain grows. This requires, at least,

Author's address: PREDA Dev Team, devteam@preda-lang.org.

that all transactions making changes to ledger states must be transferred and executed in every node to keep the ledger states correctly updated.

Such a constrained programming model makes development relatively simple, equivalent to programming a single-thread CPU and to fit everything in a single-box computer. Such a model is widely adopted and shared by most blockchain systems and smart contract languages nowadays ever since Ethereum and the Solidity language were introduced, such as Move [2] from Facebook Diem, Cadence [4] from Flow blockchain, Scilla [7] from Zilliqa, etc. It works well as long as the workload of executing all transactions and maintaining the entire ledger states fit in a single networked computer with moderate Internet connection.

1.1 Multiple Chains

Increasing DApps/addresses population and transaction volume demands higher throughput and capacity of smart contract execution and ledger state storage. However, it is capped by the computing resource a single computer may have. A temporary workaround is to accept only crazy high-end computers with insane high-speed internet connection [5], at the cost of sacrificing decentralization.

Dividing and distributing workload to multiple computers is a time-tested design philosophy to achieve the scalability of a computing system. As for blockchain systems, leveraging multiple blockchains and distributing workload of the entire network across different instances of blockchains is the fundamental solution in the long run, so that the infrastructure is able to scale continuously as the crypto ecosystem grows.

Figure 1 illustrates typical structures of multi-chain blockchain systems. Blockchain sharding employs multiple chains and one-per-shard with synchronous chain growth (a) (e.g. NEAR [9]), or asynchronous chain growth (b) (e.g. Monoxide [11]). In blockchain sharding, chains in all shards are functionally equivalent but dealing with different non-overlapped set of transactions and ledger states. All smart contracts are deployed and can be executed in every shard, while transactions, addresses and ledger states are divided and distributed in different shards with a deterministic and non-overlapping approach.

In figure 1(d), a blockchain system is distributed in a heterogeneous way with application-specific Parachains (e.g. Polkadot [12] and OHIE [13]). Each smart contract is deployed, and only deployed, in a dedicate chain (a Parachain), such that all transactions, addresses and ledger states involving the smart contract will be, and only be, handled in the corresponding Parachain. Every Parachain is unique and can be totally unrelated to other parachains in the system unless cross-contract invocations are made.

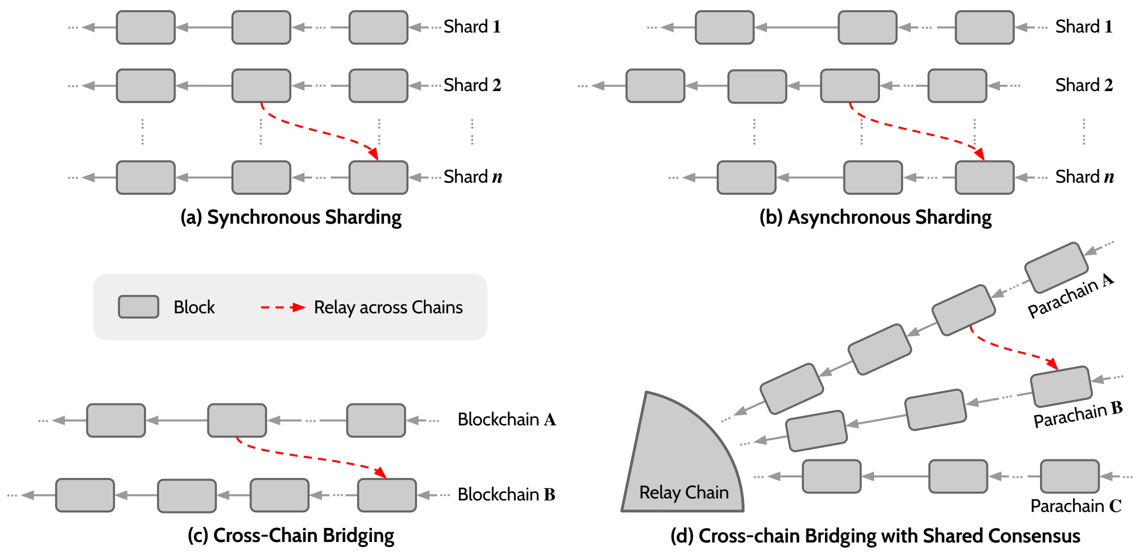

Fig. 1. Typical structures of multiple chains employed in blockchain systems.

Scalability is achieved by allowing a node to participate in only a few, typically one, shards (or Parachains) in the system and carries a fraction of the workload in the entire network for ledger state updates and transaction executions. All these nodes jointly have all shards well maintained and support the workload of the entire network. As the entire network is divided to more shards and more nodes participate into shards, the total throughput and capacity of the network scales out without limit.

Despite the scalability, interoperability of different blockchain systems is also an important scenario of dealing with multiple chains as illustrated in (c). In a cross-chain bridge, assets in one blockchain can be moved or warped to its representation across different blockchain systems with totally different set of addresses, smart contracts and programming languages. Parachains system mentioned above can also be regarded as a collection of multiple independent blockchains plus a common shared cross-chain bridging infrastructure (shown as the Relay Chain in (d)).

There are other types of multiple chain structures. Zilliqa with COSPLIT [10] splits and distributes transaction executions on different chains, but the ledger state storage is not divided. Each node on the blockchain must store all addresses and states, and a state synchronization step is required after each epoch. Prism [1] decomposes the blockchain into multiple chains based on functionalities, such as chains for block proposals, voting, final block creation, etc. The execution of each transaction goes through all chains and each blockchain node must store all states.

1.2 Functional Relay

Cross-chain payments or assets moving in a multi-chain system are essential and inevitable as discussed in many existing works [6, 11]. Such behavior can be implemented as the execution of a dual-step operation of withdraw and deposit (line 7 and 8 in Listing 1) in each involved chain respectively, which is called *Relay*, or *Relay Transaction*. Various methods have been developed to ensure the security of relay that the first step (withdraw) has been done successfully in the originate chain with a proof that can be verified in the destination chain to carry on the second step (deposit) with confidence.

PREDA generalizes the way of expressing cross-chain workflow in smart contracts to *Functional Relay*, which is programmable and flexible with the prerequisite that the security of relay across chains is ensured by the underlying consensus system. Functional relay enables any function invocation (*an initiate function*) executed on one chain to trigger one or more subsequent asynchronous invocations of functions (*relay functions*) in other chains.

A functional relay is emitted by an attempt of invoking a cross-scope function and is encapsulated as a *functional relay transaction* along with auxiliary metadata helping relay verification in the destination chain. The transaction, denoted as $\langle \mathcal{P}, \Omega_d, \lambda, \phi \rangle$, carries:

- **Relay Proof** $\mathcal{P}$: A piece of information generated by the originate chain that proves the initiative of the functional relay invocation, which can be verified in the destination chain. Typically it is a path in the Merkle tree composed by the shared consensus system like in sharding systems (a,b) and Parachain systems (d), or a digital signature by the controller of cross-chain bridges (c).
- **Target Scope** Ω_d: A context specifying the subset of the ledger states where the specified function to be executed within. It can be identified by an address that implicitly specifies the destination chain in sharding system (a,b), or an explicit identifier of the destination chain in cross-chain bridging system (c,d).
- **Function** λ: An identifier of a specific function of a specific smart contract, defined in scope Ω_d, to be executed in the destination chain, which is deployed beforehand. Note that the initiative function executed in the originate chain and the relay function executed in the destination chain can be implemented in different programming languages and executed by different execution engines as long as their type systems for function augments are compatible and inter-convertible.
- **Arguments** ϕ: The serialized data of arguments for invoking the function λ in the destination chain using the type system and execution engine in the destination chain.

Similar to the processing of normal transactions (issued and signed by addresses), functional relay transactions are issued, when a transaction is executed in the originate chain, and are transmitted to the broadcast network of the destination chain, where they reside in the mempool and wait to be confirmed by a future block of the destination chain. The major difference between this and a normal transaction is that a functional relay transaction is verified based on the relay proof rather than a digital signature of the sender.

1.3 Programmable Scopes

Scope Ω is the context of smart contract function execution, defining a collection of state variables and functions available for access and invocation. In single-box blockchain systems as described in section 1, scope is trivial. It is always the entire ledger state, including the states of all addresses (e.g. line 3 in Listing 1) and functions of all smart contracts. In many multi-chain systems, the entire ledger state is divided by address (e.g. Blockchain Sharding) or by smart contract (e.g. Parachain) to distribute the workload on multiple chains. However, such division is fixed, hardcoded and coupled with the underlying blockchain system. Due to the lack of programmability, general smart contracts cannot be implemented in a simple and readable way, which limits the applications of these systems.

PREDA generalize the basic division idea to *programmable scope*, or *scope*, with the programmability that division schemes can be defined in smart contracts by developers. A ledger state can be divided by any data types besides address, and be instantiated in customized ways. In multi-chain systems, a scope $\Omega_i \Rightarrow \langle \mathcal{S}_i, \mathcal{F}_i, \Psi_i \rangle, i \in \Phi$ is a subset of the entire ledger states with:

- **Identifier Space** Φ: The set of all possible values $\{i\}$ can be used to identify a scope, typically all addresses. It can also be all values of a particular data type such as integers, strings or hash values.
- **Ledger States** $\mathcal{S}$: A subset of non-overlapping ledger states that are read and written sequentially.
- **Functions** $\mathcal{F}$: A subset of smart contract functions that are restricted to only accessing ledger states $\mathcal{S}$ and only invoking functions $\mathcal{F}$ in the same scope. Any cross-scope access or invocation requires asynchronous functional relay.
- **Chain** Ψ: A chain that hosts states $\mathcal{S}$ and sequentially executes transactions carrying invocations of functions in $\mathcal{F}$.

Different scopes never share states $\mathcal{S}$ and may share entire, partial or no functions $\mathcal{F}$ of another scope in various designs of multi-chain systems, which implies different programming flexibility, synchronization mechanisms and scalability. Different scopes may also share the same chain Ψ, or have their own dedicate ones. The logic behavior in a scope can be completely defined with states $\mathcal{S}$ and functions $\mathcal{F}$, while we factor the chain Ψ into the formula though, so that the performance of cross-chain smart contracts can be better analyzed. Any functional relay across chains introduces a communication overhead and mempool awaiting delay that can not be ignored.

Table 1 lists division schemes of scopes in typical multi-chain systems. In blockchain sharding, smart contracts are not divided. Homogeneous scopes are instantiated for each scope identifier based on the same definitions of states and functions, and so do shards, which achieves best scalability that each smart contract is possible to leverage the throughput and capacity of all chains in the entire network, but has most narrowed scope. In Parachain system, each scope is corresponding to a specific smart contract, which is heterogeneous that each scope has unique definition of ledger states and functions by the contract. Such design exhibits better programming flexibility with contract level scope but restricts scalability by limiting the throughput and the capacity of a single chain to be utilized. PREDA model provides programmability of the scope division scheme, which can be customized and optimized according to the data access pattern and the predicated runtime behavior of cross-chain

	Ledger States	**Smart Contracts**	**Chain**
Sharding System	Homogeneous Instance per-Address Homogeneous Instances per-Shard	No Division	per-Shard
Parachain System	Heterogeneous Instance per-Contract		per-Contract
Cross-chain Bridge	Heterogeneous Instance per-Chain		per-Chain

Table 1. Division of Ledger states, smart contracts and broadcast network in typical multi-chain systems.

workflow of smart contracts. Customized scope division enables better trade-off between programming flexibility and scalability.

2 PROGRAMMING MODEL

PREDA programming model is a distributed, functional, scope-oriented and high-level approach for defining and implementing inherently-parallelized smart contracts that runs on multi-chain blockchain systems. Ledger states and functions are defined within scopes, each is hosted by an underlying chain. Cross-scope interactions are described using *functional relay semantics* that ensures the availability of context of any function invoked across chains asynchronously. *Programmable scope schemes* provides a systemic and expressive way to design the division scheme of ledger states of a smart contract that can be transparently distributed with inherent parallelization by the underlying multi-chain system.

2.1 Scope-Oriented Smart Contracts

In each smart contract, its states $\mathcal{S}_i$ and functions $\mathcal{F}_i$ are defined within a scope Ω_i ($i \in \Phi$) as described in section 1.3. Any function allows direct access of states and synchronous invocation of functions only within the current scope. Any reading/writing of states, invocation of functions in another scope Ω_j must to be realized using functional relay in an asynchronous manner as described in section 1.2.

Each scope is allowed to have its own definition of states and functions, which is typically the case of cross-chain bridging system and Parachain system. While for blockchain sharding systems, scopes can also be instantiated per scope identifier i based on a declaration of *scope class* as $\langle \mathcal{S}, \mathcal{F} \rangle$. Every scope instantiated from the same scope class will have the same set of functions and same state data structure, but with possibly different values. Given a scope identifier space Φ, all scopes derived from the scope class $\langle \mathcal{S}, \mathcal{F} \rangle$ are,

$$\Omega_i \Rightarrow \langle \mathcal{S}, \mathcal{F}, \Psi(i) \rangle, \quad i \in \Phi. \tag{1}$$

Here, *identifier scatter function* $\Psi(i)$ is an analytic and deterministic mapping from scope identifier to the index of a particular chain in a multi-chain system, assuming chains are named by integer numbers as index.

For example, scopes of all addresses can be represented by a scope identifier space Φ of all addresses (e.g. 20-byte hash of public key in Ethereum), which is typically an ad-hoc predefined fixed states division hardcoded in the underlying sharded blockchain system. PREDA model enables programmable definition of schemes of ledger state division identified by more data types besides address, and makes state division decoupled with the design of the underlying blockchain system.

2.2 Distributed Scopes

For state storage and transaction processing, a scope is entirely located in one, and only one specific chain Ψ, which is determined by the identifier scatter function $\Psi(i), i \in \Phi$ for sharding systems. PREDA model doesn't define such a function and leaves the definition to the underlying blockchain system, which is usually coupled with the configuration, architecture and data format of the multi-chain system. The identifier scatter function

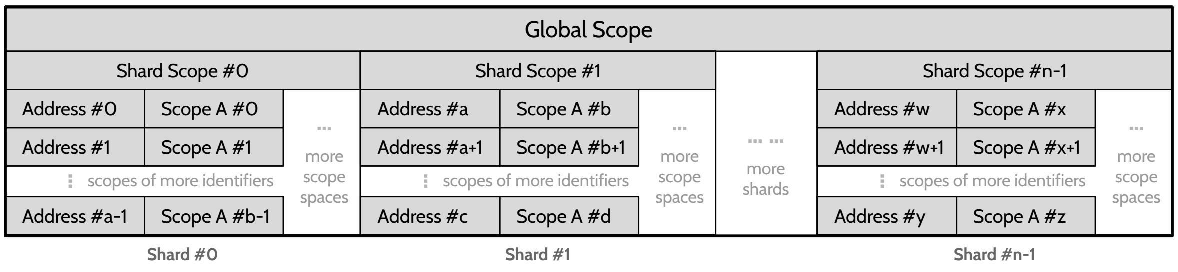

Fig. 2. Entire ledger states divided by programmable scopes in a sharded blockchain system. Each block represents a scope built-in (`Shard` and `Global`), or instantiated based on scope space defined in smart contracts (`Address` and `ScopeA`) with various types of scope identifier. Instantiated ones are scattered throughout all shards.

$\Psi(i)$ is required that the function can be analytically evaluated without heavy computation nor relying on any allocation/assignment service. It is highly recommended that an identifier scatter function has high expectation to have a balanced distribution across chains. Using truncated hashing on data representation of scope identifiers i can be the typical solution in many cases. In cross-chain bridges, identifier scatter function $\Psi(i)$ is unnecessary since scope identifier i is also the identifier of blockchains involved in the bridging system.

Address Scopes are defined by lettering Φ be all possible address values in the network, which is frequently used. In PREDA model, it is recommended to have address scope built-in and predefined, which achieves better compatibility with existing multi-chain systems based on ad-hoc per-address states division architecture.

Shard Scopes $\ddot{\Omega}_s$ can be defined by lettering Φ be the set of all shard indices s in blockchain sharding systems. In PREDA model, these scopes are built-in and predefined, but the expressiveness of referring to a specific shard is disabled. PREDA model aims to minimize the exposure of details of the sharding structure (e.g. total number of shards or index of the current shard.) so that a smart contract can be ported to different sharding systems and adapted to dynamic chain scaling without code modification. Similarly, *Chain scopes* can be defined by lettering Φ to the set of identifiers of blockchains/Parachain for cross-chain bridging systems. Explicitly referring a specific chain in cross-chain bridging systems is allowed and necessary. We refer any scopes derived from equation 1 excluding shard scopes as *conventional scopes*.

Global Scope $\widehat{\Omega}$ is a special built-in scope, which is a logically global singleton in the entire network. Exceptional to the above discussion about programmable scopes, global scope is not a division of the ledger states, instead, it has a fully duplicated instance maintained by every chain, and synchronized throughout the entire multi-chain system. Global scope carries states that must be available to all chains and all scopes. Transaction processing for global scope can not be scaled nor parallelized, which suggests only necessary ledger states should be defined in global scope and minimize transaction traffic involving global scope.

2.3 Inherent Parallelization

Every chain in a multi-chain system is logically a sequential state machine. Transaction executions of each block drive ledger states transition by sequential invocation of smart contract functions carried by these transactions. In a multi-chain system, each sequential state machine (a chain) operates independently, and jointly forms a parallel computing system with message-passing (relay transactions) for inter-thread coordination.

PREDA model provide no explicit primitives for local multi-threading or distributed parallelization. Instead, it leverages existing parallelism of the multi-chain system by distributing ledger states and transactions across these chains. The case of blockchain sharding system is illustrated in figure 2. Scopes distributed in different shards are operated in parallel inherently (e.g. `Address#0` and `Address#a`), while transactions involving a specific scope should still be executed sequentially.

Intra-chain Parallelism with local multi-core processors can also be leveraged besides the cross-chain parallelization, by creating multi-threaded workers for transaction execution. Scopes can be scattered to these local threads to transparently accelerate the execution of a block carrying a considerable number of transactions, which achieves local parallelization without bringing additional complexity to the programming model. Identifier scatter function $\Psi(i)$ can be reused here for distribution across local thread, by referring to thread index instead of shard index.

2.4 Functional Relay

Any function invocation across chain must be facilitated by a functional relay transaction $\langle \mathcal{P}, \Omega_d, \lambda, \phi \rangle$ as described in section 1.2, even if the invocation across scopes are in the same contract, or the two scopes are actually distributed in the same chain. On the other hand, PREDA model allows direct states access and function invocation across smart contracts within the same scope.

PREDA model provide primitives to make asynchronous invocation by specifying the destination scope, function and function augments as $\lambda(\phi; \Omega_d)$ for conventional scopes and $\lambda(\phi; \widehat{\Omega})$ for the global scope. When transaction execution reaches such a primitive, a functional relay will be emitted and collected. After all transactions in the block are executed, all collected functional relay will be encapsulated as functional relay transactions. Each one will be dispatched to the destination scope and confirmed there to realize the function invocations in the destination scope. As illustrated in figure 3, a functional relay transaction undergoes following steps to complete an asynchronous invocation across scopes in different chains:

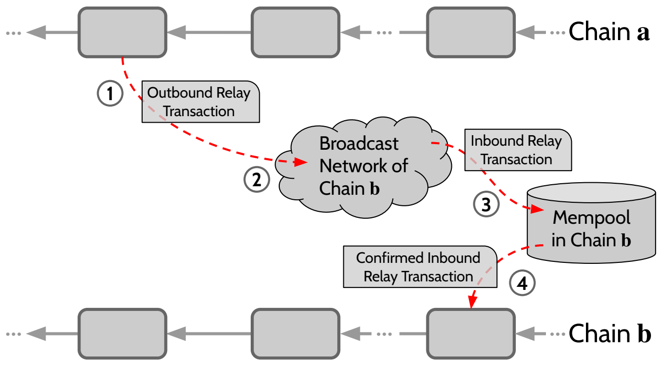
Chain a
1
Outbound Relay Transaction
Broadcast Network of Chain b
2
Inbound Relay Transaction
3
Mempool in Chain b
Confirmed Inbound Relay Transaction
4
Chain b

Fig. 3. Workflow of a functional relay transaction emitted from the originate chain a and confirmed in the destination chain b.

(1) A functional relay transaction is emitted during block execution in the originate chain a, as *outbound relay transaction.*
(2) Underlying system transfers the transaction to any node in the broadcast network of the destination chain b, which is identified by $\Psi(d)$.
(3) A node in the destination broadcast network receives the functional relay transaction, as *inbound relay transaction*, and stored in the mempool of chain b. The inbound relay transaction awaits there to be picked by a future block in chain b.
(4) Finally, a new block in chain b confirmed the inbound relay transaction and concludes the workflow.

PREDA model reuses existing modules (the broadcast network and the mempool), which are supported in most multi-chain systems, to realize the workflow by introducing a few new metadata in transaction data structure and additional steps to transaction processing. PREDA model rely on the underlying consensus system to generate the relay proof $\mathcal{P}$ at the originate chain and verify it at the destination chain. In programming model level, it is assumed that relay proof generation and verification is transparently and correctly handled.

Functional Relay Broadcast $\lambda(\phi; \{\ddot{\Omega}_s\})$ [1] is a dedicate primitive for invoking the same function with the same augments in every shard scope without explicitly specifying a particular shard. In blockchain sharding systems, the destination scope in a functional relay can be the global scope or any conventional scopes except shard scopes. It is not allowed to explicitly specify a shard scope as the destination of a functional relay since PREDA model intends to hide the details of sharding structure from programming level. If invocations of a function in every shard scope are desired, the functional relay broadcast serves the purpose. Similar to functional relay, functional relay broadcast can be emitted from any scope including shard scopes. A functional relay broadcast is logically equivalent to emitting multiple functional relay transactions per-shard with the same invocation parameters.

As a simple example, listing 2 illustrates the code snippet of a smart contract defining a token and the transfer function with PREDA model. Pseudo-codes of `scope` and `relay` are primitives to define a scope class (line 3-16) and to emit a functional relay (line 10-13). Line 10-13 also defines a Lambda function with `amount` argument captured to perform the deposit in the destination scope. Scopes will be instantiated for every address as formulated in equation 1, based on the definition in line 3-16. Each has a variable `balance` and a `transfer` function with a Lambda function embedded. The execution is initiated in the scope of sender's address (`@msg.sender`, implicitly specified), which attempts a withdraw. If succeeded, the relay primitive (line 10) emits a functional relay and concludes the execution in sender's scope. Then, asynchronously in recipient's scope (`@receiver`), the Lambda function will be invoked and completes the workflow with a deposit.

[1] $\{\ddot{\Omega}_s\}$ is a fixed notation to indicate the broadcast behavior instead of specifying an actual destination scope.

```
contract ERC20Basic is IERC20
{
    scope @address
    {
        uint256 balance;
        function transfer(address receiver, uint256 amount) ...
        {
            require(amount <= balance);
            balance = balance - amount;
            relay @receiver (^amount)
            {
                balance = balance + amount;
            }
            return true;
        }
    }
}
```

Listing 2. The code snippet of `transfer` function in an ERC20 contract rewritten in extended Solidity with PREDA model

2.5 Colocated Scopes

As mentioned in section 2.4, invocation across scopes are facilitated by cross chain relay transactions because the target chain (where the target scope is located) might be different from the source chain and the node is not participating in it. If it is guaranteed that any node participating in the source chain also participates in the target chain, it would enable an optimization, that the actual relay processing workflow be bypassed and replaced with direct invocation by the execution layer of the chain, because the target scope is accessible within the same node. In such a case, we call the target scope a *colocated scope* of the source scope.

As described in section 2.2, shard scope and global scope have guaranteed colocation with conventional scopes by design, which can be categorized based on how such colocation is shared:

- **Synchronous Shared Colocation**. The shard scope of a chain is a colocated scope of every conventional scope that is located in that chain. Since it's within the same chain as those conventional scopes, they are operated synchronously.
- **Asynchronous Shared Colocation**. Because the global scope requires all nodes in the multi-chain system to participate in, it is a colocated scope of every other scope. It is hosted by a dedicated *global chain*, so the other scopes and the global scope are operated asynchronously. (Section 3.3)

PREDA model allows direct function invocation and states access (read/write) from a scope to another scope that is colocated synchronously shared. While, for asynchronous shared colocation, direct invocation is restricted to const functions and direct states access is read-only. Complete details of direct access across scopes are listed in Table 2. In blockchain sharding system, only functional relay broadcast is allowed toward an invocation in shard scope, denoted as **R/b**.

Originate	Destination				
	Global		Shard		Conventional
	Read	Write	Current	Foreign	
Global	-		-	R/b	R
Shard	D	R	-	R/b	R
Conventional	D	R	D	R/b	R

† **D**: direct access/invocation is allowed, **R**: a functional relay is required.
‡ **R/b**: only a functional relay broadcast is allowed without specifying any specific shard.

Table 2. Rules for state access and function invocation across different scopes according to different situations of scope colocation. **Conventional** refers to all scopes derived from equation 1 excluding shard scopes.

Allowing direct writing and invocation of non-const function to colocated shard scope will break the independency of scopes within a chain. The scope dependency assumed by intra-chain parallelization described in section 2.3 will be violated. Thus, intra-chain parallelization can only be applied to a set of transactions without direct write access to the shard scope.

3 HOSTING MODEL

Smart contracts based on PREDA model execute on multi-chain systems with fixed number of chains (cross-chain bridging), configurable number of chains (blockchain sharding) or infinite number of dynamically allocated chains (Parachain). These chains are assigned with fixed names as labels, consecutive integers or hash values, a.k.a. *chain identifiers* which is agreed on and recognized by both PREDA model and the underlying system.

For every chain in the multi-chain system, PREDA model makes similar assumptions to a single-box blockchain system (e.g. Ethereum). Every chain has following essential components regardless of the actual consensus algorithm employed:

- **A Chain of Blocks** generates new block periodically with fixed, or fixed expectational, interval. Each block carries an ordered listed of unique transactions with limited total data sizes, or total computation cost for execution (e.g. Gas).
- **A Storage for Ledger States** provides efficient immediate read and write of state data.
- **An Execution Engine** runs deployed smart contracts as invocations made by transactions and updates ledger states according to execution outputs.
- **A Mempool** stores, in memory, unconfirmed transactions to be picked up by future new blocks.
- **A Broadcast Network** is a peer-to-peer network that replicates legitimate transactions and blocks across nodes.

In a multi-chain blockchain system, each chain has its own unique dedicate instances of the five components described above. In addition, PREDA model makes following assumptions about a multi-chain system:

- **Non-overlapping Workload**: Any transaction will be, and only be, confirmed and executed by a single chain. An address or any piece of ledger state will be, and only be, hosted and updated by a single chain.
- **Deterministic Distribution**: A transaction, an address or any piece of ledger state will always be associated with a specific chain, given a particular configuration of the multi-chain system.
- **Proven Relay**: A relay proof can be generated from a block of one chain, and be verified by another chain when the corresponding relay transaction is received.
- **Voluntary Relay Broadcast**: A relay transaction with correct proof will be replicated across nodes and stored in mempool voluntarily like normal transactions being done.

3.1 Execution Engine

PREDA model assumes a state-less execution engine like EVM, which executes smart contract functions, reads the ledger states as constant and produces a collection of modified states without directly writing to the ledger states. An execution context is exposed to the function as a runtime library, which provides ledger states access, relay emission and auxiliary information from the transaction/block being executed like the block height and the address of transaction sender.

Interfaces of the execution engine has immediate interaction with the PREDA model, and subject to a few modification to reflect the new data model of the divided ledger states with programable scopes and the new behavior of emitting functional relay transactions.

The ledger states access requires a scope identifier i as an addition to the pair of a smart contract address s and a variable location v. The variable location v will be specific to the combination of the scope and the smart contract $\langle s, i \rangle$, instead of just to the smart contract s. In a function of a smart contract, variables are referenced without specifying the scope identifier, instead, the current scope is implied. This is a typical behavior in object-oriented programming models like the *this* pointer in a C++ object, which requires the current scope identifier be part of the execution context. Listing 2 line 9 and 12 illustrates such behaviors as an example. The state variable (`balance`) is referenced without explicitly specifying an address, which is actually implied as the current scope (`@msg.sender` in line 9 and `@receiver` in line 12).

A new interface shall be introduced for functional relay emission by specifying the destination scope identifier d, the function λ, its augments ϕ and a gas fee redistribute weight ρ. Implementation of the interface is required to collect all relay emissions, encapsulate each to a relay transaction and forward them to the broadcast network of the destination chain. Forwarding of relay transactions can be asynchronous and is tolerate to delay. The gas fee redistribute weight ρ determines how much gas fee will be offered to the relay transaction to be emitted, which divides up the residue $\hat{g}$ of the gas fee after the current execution. Actual gas fee g_r of a relay transaction r out of total b relay(s) being emitted is determined as

$$g_r = \hat{g} \cdot \frac{\rho_r}{\sum_{0 \le x < b} \rho_x} \tag{2}$$

3.2 Transactions and Blocks

Besides information about the invocation (the function λ and its augments ϕ) and the metadata like gas price/limit, a transaction carries the digital signature by the sender, or by the controller of the cross-chain bridge. With PREDA model in blockchain sharding systems, a transaction shall be able to alternatively carry a relay proof for authenticity other than a signature.

PREDA model requires the destination scope identifier to be carried with relay transactions in blockchain sharding system, while the destination scope of a normal transaction can be derived from the public key in the signature data. The destination scope is implied for cross-chain bridge based on the blockchain it is sent to. Parachain system supports such information already for relay transactions.

A block carries an ordered list of confirmed transactions. In addition to normal transactions, confirmed relay transactions will be included as well, which forms an ordered list of normal/relay transactions mixed. A block carries an aggregated proof (e.g. a Merkle tree root) of all transactions being confirmed in most blockchain systems. In addition to that, a block may also carry an aggregated proof for all relay transactions emitted when executing all transactions confirmed in the block.

3.3 Global Scope

Global scope is optional but particularly useful to aggregate and publish information, which involves or to be made available to all programable scopes and all chains in system, for example, collecting votes and deploying smart contracts in a blockchain sharding system. Global scope is a special scope that requires all nodes in the entire multi-chain system to participate in to maintain ledger states and execute transactions of global scope. It can be hosted by a dedicate chain, *global chain*, running on every node in parallel to the existing multi-chain system, which makes global scope available to any programable scopes. Besides the different node participation model, the global chain works exactly in the same way as existing sharded chains.

As discussed in section 1.1, synchronous sharding (figure 1a) will have an additional shard dedicate for global scope in parallel to existing shards and provide a synchronized consistent view of the global scope across all nodes in the entire network. Appendix A presents a reference design of such a synchronous sharding system with consistent view of global scope. Similarly asynchronous sharding (figure 1b) also have such a global chain while a consistent view cannot be guaranteed due to its asynchronous nature of sharded chain growth.

4 CONCLUSION

This article presented a novel programming model, PREDA, for the development of general smart contracts on multi-chain blockchain systems, especially for homogeneous sharding systems. PREDA model divides the sequential state machine of the entire ledger state into a great number of independent sequential state machines, *scopes*, which can be arbitrary distributed in any chain in a multi-chain system to leverage the throughput and capacity of all chains. Scope-based division decouples the design of a distributed smart contract from the underlying multi-chain architecture, and ease the development of general smart contracts on parallel multi-chain systems.

PREDA model presents a scope-oriented programming paradigm with *programmable scope schemes* to control state division and define the actual sequential state machine within each scope, and *functional relay semantics* to describe the cross-scope workflow to facilitate the interaction and coordination across scopes. PREDA model is neutral to different types of consensus algorithms and architectures of multi-chain systems and can be applied to sharded blockchain systems as well as Parachain systems and cross-chain bridges.

Appendix A SYNCHRONOUS SHARDING WITH GLOBAL SCOPE

A synchronous blockchain sharding system works best with the proposed PREDA model, which provides full features with globally synchronized consistent view of global scope at every block height. A reference design of such a system is illustrated in figure 4 by extending a single-chain architecture that most nowadays blockchain systems have.

A.1 Multi-Chain Structure

The existing single-chain as well as the ledger states, execution engine, mempool and the broadcast network will all serve transactions in global scope only, named as g. The system is then extended by allocating additional 2^k *sharded chains*, named as 0 to $2^k - 1$, with their own dedicate instances of ledger states, execution engine, mempool and the broadcast network which handle transactions of programable scopes distributed in each chain only. Parameter k is the *sharding order* controls the overall size of the sharding system, exponentially, which makes total number of chain to be integral power of two. The identifier scatter function $\Psi(i)$ discussed in section 2.2 can thus be simply taking first k bits from the hash of scope identifier and achieve a good balanced distribution across chains.

As a synchronous sharding system, for each new block generated in global chain, one, and only one, block per sharded chain will be generated, which results in aligned block heights for all chains in the system. In any node participated in one or more sharded chain(s), the block of global chain at height h shall be received and executed after any block at height $h - 1$ is executed and prior to any block at height h in any sharded chain, which provides a consistent view of global scope at height h throughout the entire network when executing any block in sharded chains.

A.2 Data Structures

A sharded chain doesn't have own consensus proof, instead it inherits consensus proof from the global chain. Figure 5 illustrate key data structures that extend a single-chain blockchain system with sharded chains. Existing data structures, block header and block body, of the single-chain system are denoted here as *consensus header* and *global block*. The consensus header carries the proof for a validated consensus proof (e.g. the PoW nonce, or the aggregation of PoS signatures) and the hash pointing to the global blocks θ_g, which carries actual transactions in global scope being confirmed. The two data will be broadcasted in the global broadcast network that all nodes in the network will receive those regardless of the sharding division. Every node thus has the ledger states of the global scope and keeps updated.

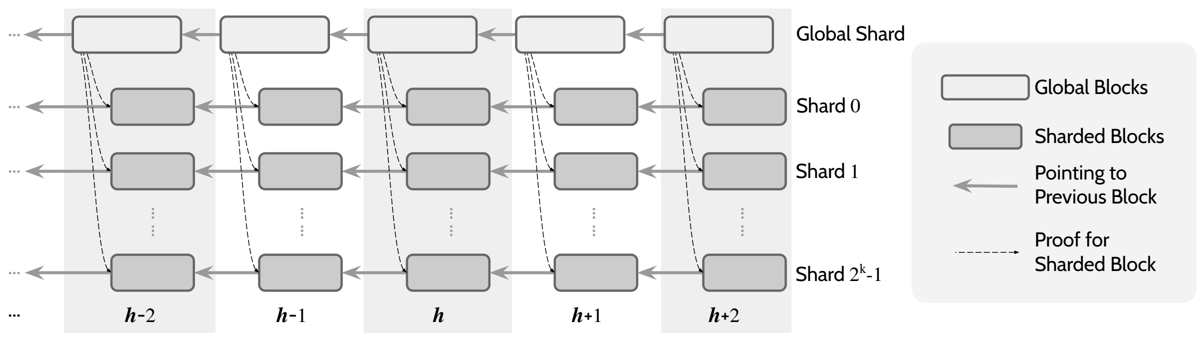

Fig. 4. A reference design of synchronous blockchain sharding system with the global scope.

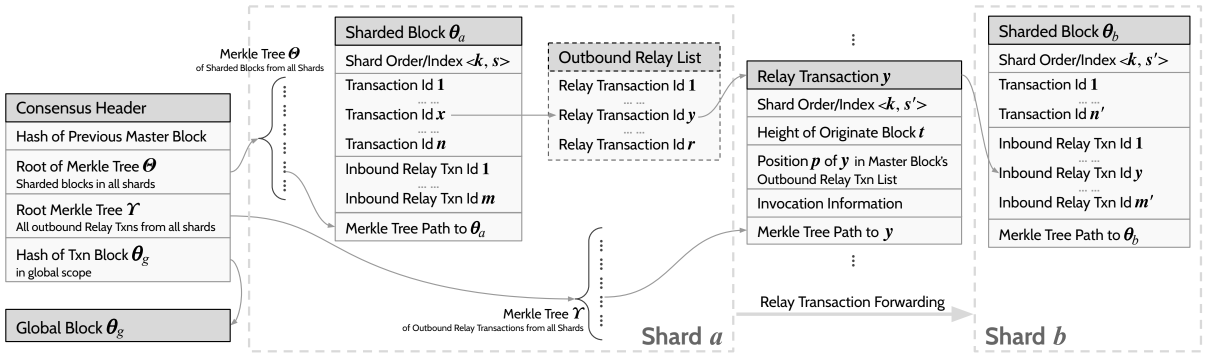

Fig. 5. Metadata in block data structures for the workflow of a functional relay.

To extending the global chain with sharded chains, two additional aggregated proofs (Θ and Υ) are introduced and embedded in the consensus header at every block height to prove validities of all newly generated sharded blocks and emitted relay transactions at that height h.

- A Merkle tree Θ is built by taking hashes of 2^k sharded blocks at height h of all chains. The Merkle tree root will be embedded in the consensus header so that a sharded block θ_s can be verified in any sharded chain.
- A Merkle tree Υ is built by taking hashes of relay transactions emitted by blocks at height h of all sharded chains to facilitate functional relays. Embedding the root of the Merkle tree Υ in every consensus header enables validation of any inbound relay transactions received in the global chain or in any sharded chains, by checking upon the Merkle root carried by the consensus header at the emitted block height of a particular relay transaction.

A.3 Scalability

Increasing number of shards expends throughout and capacity of the entire network linearly. The additional overhead carried in every node rises up as well. With total $n = 2^k$ sharded chains, following overhead of data broadcast is introduced:

- **Additional Merkle Tree Roots** add 32×2 bytes [2] to the consensus header, while it is a tiny constant overhead independent of n.
- **Sharded Block Proof** is a path in Merkle tree Θ and adds $32 \log_2 n$ bytes to each block, which is a sub-linear overhead as the number of shards n grows.
- **Relay Proof** carried by every relay transaction is a path in Merkle tree Υ and adds a sub-linear overhead of $32 \log_2 (m \cdot n)$ bytes to each relay transaction. Here, m is the average number of functional relays emitted by each block, which is roughly constant as well.

Sharding introduce no overhead to the storage of ledger states and negatable computation cost for Merkle root reconstruction and comparison. In summary, only logarithm sub-linear overhead is added in every node with increasing number of shards, which allows the presented architecture well linearly scaled.

REFERENCES

[1] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. 2019. Prism: Deconstructing the Blockchain to Approach Physical Limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. Association for Computing Machinery, 585–602.

[2] Assuming SHA256 is employed for building the Merkle tree. Same for further items.

[2] Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. 2020. Move: A Language With Programmable Resources. https://diem-developers-components.netlify.app/papers/diem-move-a-language-with-\programmable-resources/2020-05-26.pdf.
[3] Vitalik Buterin and et al. 2013. A Next-Generation Smart Contract and Decentralized Application Platform. https://github.com/ethereum/wiki/wiki/White-Paper.
[4] Cadence Developers. 2020. Introduction to Cadence. https://developers.flow.com/cadence.
[5] Solana Foundation. 2022. Validator Requirements (Solana Documentation). https://docs.solana.com/running-validator/validator-reqs.
[6] Eleftherios Kokoris Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *Security and Privacy (SP), 2018 IEEE Symposium on*. Ieee.
[7] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer Smart Contract Programming with Scilla. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 185 (oct 2019), 30 pages.
[8] Ethereum Dev Team. 2021. Solidity Documentation. https://docs.soliditylang.org/en/latest/.
[9] The NEAR Team. 2022. The NEAR White Paper. https://near.org/papers/the-official-near-white-paper/.
[10] The Zilliqa Team. 2017. The ZILLIQA Technical Whitepaper. https://docs.zilliqa.com/whitepaper.pdf.
[11] Jiaping Wang and Hao Wang. 2019. Monoxide: Scale Out Blockchain with Asynchronous Consensus Zones. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*. USENIX Association, 95–112.
[12] Gavin Wood. 2017. Polkadot: Vision for a Heterogeneous Multi-chain Framework. https://polkadot.network/PolkaDotPaper.pdf.
[13] Haifeng Yu, Ivica Nikolić, Ruomu Hou, and Prateek Saxena. 2020. OHIE: Blockchain scaling made simple. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP '20)*. IEEE, 90–105.

The PREDA Language

A Distributed Programming Language for General Smart Contracts on Sharded Blockchains and Cross-Chain Bridges

PREDA DEV TEAM

PREDA, (**P**arallel **R**elay-and-**E**xecution **D**istributed **A**rchitecture), is a novel programming model for general smart contracts dealing with multi-chain blockchain systems, in which logic execution and ledger states are partitioned and distributed across chains. In this document, we introduce the core features of the PREDA programming language with code examples.

The PREDA language is a high-level programming language with curly-bracket and Algol-like syntax style for general smart contract development. It is naturally parallelized and distributed based on the proposed programming model, which provides object-oriented scope syntax and lambda function syntax to support the two core features of the PREDA model. The PREDA language aims to maximize the interoperability with sequential, non-distributed programming languages (e.g. Solidity), which allows cross-language invocation with unifies type system and asynchronous calls to/from languages with sequential programming model.

Compilation system and execution engine of the PREDA language is implemented as compiling to intermediate representation, and then building to platform-specific native binaries or virtual machine bytecodes. Solidity compiler and Ethereum Virtual Machine are integrated as parts of the system to realize the interoperability and cross-language compilation. Compilation system generates function symbols and unified argument serialization, which enable cross-language invocation and relay transaction composition based on functional relay semantics.

1 SCOPES AND RELAYS

The PREDA programming language provides syntax to decompose states of a smart contract into **scopes** with different synchronization requirements within- and across-chains/shards, which enables state sharding for general smart contracts rather than payment contracts. The PREDA language also provides syntax to decompose execution flow within a contract function into multiple **microfunctions** that are relayed and executed in different shards based on their data dependencies of states in different scopes.

1.1 Built-in Scopes

PREDA has three built-in scopes, i.e., **global**, **shard** and **address**. They correspond to the data partitioning structures in sharded blockchains. Programmers specify the scopes of the contract variables, and the underlying blockchain system initiates and executes contract functions on the scopes accordingly:

- **Global scope** is used to define common contract states that must be shared and synchronized by all shards. A global scope state is effectively a singleton that is available throughout the network. In addition, all deployed contracts and their codes belong to the global scope. A contract can import and interpret any function from another deployed contract.
- **Shard scope** is used to define common contract states within a shard, but not specific to a user. A shard scope state must be instantiated and updated independently in each shard.
- **Address scope** is used to define contract states for each user. An address scope state is initiated and updated individually in each address. This is the finest granularity a contract can expose to the underlying system.

Author’s address: PREDA Dev Team, devteam@preda-lang.org.

From the perspective of object allocation, the global scope is the equivalent of a conventional smart contract. Everything defined in the global scope has an instance on all blockchain nodes and the instances are consistent globally. The shard scope defines a state that has one instance on each blockchain node of the shard and the instances are consistent in the shard. The address scope defines a state that has one instance for an address and only blockchain nodes belonging to the same shard that have the address have the instances.

The keywords **"@global"**, **"@shard"**, and **"@address"** are used before variables and functions to define their scopes as below. Note that when a state variable or function is defined without specifying a scope, it defaults to **"@global"**.

```
contract MyContract {
    @global  uint32  numTotalAccounts;                      // only one instance globally
    @shard   uint32  numAccountsInShard;                    // one instance per shard
    @address uint512 addressBalance;                        // one instance per address
}
```

Listing 1. Three built-in scopes in PREDA

1.2 User-defined Scopes

In addition to the built-in scopes, programmers can define their own scopes as **"@scopeName = global by typeName;"** . It means that there is a user-defined scope named "scopeName", which is indexed by "typeName" on chain. The "typeName" can be one of the following by-default datatypes in PREDA: uint16, uint32, uint64, uint128, uint256, uint512, and address. Note that even when two scopes are defined with the same "typeName", they are regarded as different scopes. A user-defined scope can be used after it's defined in a PREDA program as below:

```
contract C {
  @myScope = global by uint32;
  @myScope string myStr;
}
```

Listing 2. User-defined scopes in PREDA

In this example, the scope **"myScope"** can have up to $2^{32} - 1$ instances that are indexable by a uint32 value. Which shard each of these instances resides in is decided by the underlying blockchain system and transparent to the contract.

1.3 Relay

When a contract function executing in a scope needs to access data defined in another scope that the function cannot directly access, a relay is required in PREDA. This means the transaction execution context is switched from one scope to another but the execution logic is continued. Listing 3 shows the PREDA implementation of the token transfer contract, which is equivalent to the Solidity function `transfer`. The keyword **"@address"** is used to define the scope of contract variables and functions that belong to an address. The `transfer` function executing at the scope of the sender address does the withdraw on the balance of the sender and then uses a relay to call the `deposit` function, as the deposit is executed at the scope of the recipient address.

```
contract Token {
    @address bigint balance;
    @address function bool transfer(address to, bigint amount) export
    {
        if(balance >= amount)
        {
            balance -= amount;
            relay@to deposit(amount);
            return true;
        }
        return false;
    }

    @address function bool deposit(bigint amount)
    {
        if (amount <= 0) return false;
        balance += amount;
        return true;
    }
}
```

Listing 3. Corresponding code snippet of `transfer` function in PREDA for ERC20 transfer in Solidity

As shown in this example, **"relay"** is the keyword to specify a relay statement and **"@"** is another keyword to specify the destination of a relay. A formal representation of a relay is:

$$relay@address \quad function(parameters) \tag{1}$$

Comparing the code example in Listing 3 with the Solidity implementation of ERC20 token transfer contract [3], one can find that the PREDA implementation converts the map `balances` in the Solidity implementation to an integer variable `balance` and does not specify whose `balance` is as Solidity did, e.g., message sender's balance as `balances[msg.sender]`. This is because the global variable `balances` defined in Solidity is converted to an address variable `balance` in PREDA. The execution context is switched from the sender's address scope to the recipient address scope with the relay statement.

Note that although PREDA doesn't specify how to implement a relay in the underlying blockchain system, a relay is recommended to be implemented as an asynchronous relay transaction. That is because (1) transaction is the finest granularity of communication between blockchain nodes and almost all core modules of a blockchain, e.g., transaction submission and propagation, block generation and verification, security mechanisms, etc., are built upon transactions; and (2) in a blockchain, it is unknown when a transaction will be executed. In the worst case, a transaction is discarded by blockchain nodes after an extremely long wait. The semantics of a relay must be asynchronous.

Figure 1 shows how the PREDA model executes transactions in parallel. Assuming there are four shards A, B, C, D, and two user-initialized transactions φ and φ', which can be the transactions invoking the PREDA function `transfer` shown in Listing 3. The withdraw operation ρ that only involves the state in shard A is handled by a miner in shard A. If the account balance satisfies the cost of this withdraw operation, a block $i+1$ carrying the transaction φ is created by the miner and appended to the chain of shard A. After that, a relay transaction carrying the deposit operation ϕ is composed in shard A and forwarded to shard B. The deposit operation ϕ that only involves the state in shard B can be executed by shard B. Once the relay transaction is picked up by another miner in shard B, operation ϕ is executed, concluding the complete of the transaction φ. Similarly, the withdraw operation ϕ of transaction φ' is executed in shard C and its deposit operation ρ is executed in shard D. The withdraw operations on addresses in different shards can be executed in parallel and similarly to the deposit operations.

1.4 Relays between Shard and Global Scopes

In the PREDA token transfer example (shown in Listing 3), the relays occur between address scopes. In this subsection, we shown an example that the relays occur between the shard and global scopes. Listing 4 shows the PREDA Ballot contract. It is functionally equivalent to the Solidity Ballot contract, but is designed to execute a large number of voting transactions in parallel on a sharded blockchain. The PREDA implementation is quite complex, but shows many features, e.g., relay transactions across different types of scopes. The main ideas of the

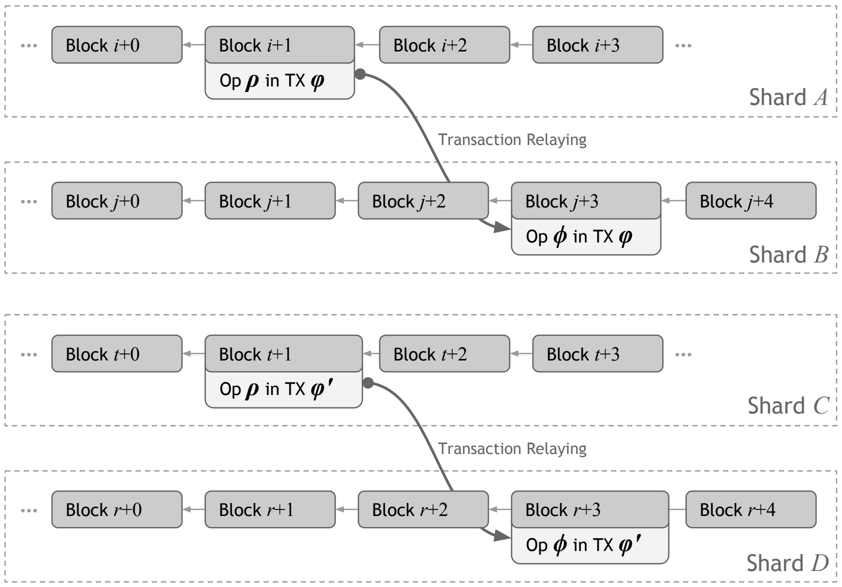

Fig. 1. Parallel execution of transactions φ and φ' in different shards.

PREDA Ballot contract can be explained as below: first, the contract allows the voters to vote in different shards based on address-to-shard mapping (by the underlying blockchain); and second, the contract accumulates the voting results of each shard to the final voting result. The PREDA Ballot contract can avoid reading and writing the shared contract state (i.e., the final voting result) simultaneously throughout the network, when processing voting transactions issued by the voters. In this case, we show how voting is implemented in the shard scope and how the final voting result is accumulated in the global scope.

```
contract Ballot {
    struct Proposal {
        string name;
        uint64 totalVotedWeight;
    }

    struct BallotResult {
        string topVoted;
        uint32 case;
    }

    @global address controller;
    @global uint32 current_case;
    @global array<Proposal> proposals;
    @global BallotResult last_result;

    @global uint32 shardGatherRatio;
    @global function shardGather_reset(){ shardGatherRatio = 0u; }
    @global function bool shardGather_isCompleted(){ return shardGatherRatio == 0x80000000u; }
    @global function bool shardGather_gather()
    {   shardGatherRatio += 0x80000000u>>__block.get_shard_order();
        return shardGatherRatio == 0x80000000u;
    }

    @shard array<uint64> votedWeights;
    @address uint64 weight;
    @address uint32 voted_case;

    @address function bool is_voting()
    {
        return last_result.case < current_case;
    }

    @address function init(array<string> names) export {
        relay@global (^names){
            for (uint32 i = 0u; i < names.length(); i++) {
                Proposal proposal;
                proposal.name = names[i];
                proposal.totalVotedWeight = 0u64;
                proposals.push(proposal);
            }
            current_case++;
            last_result.case = 0u;
            last_result.topVoted = " ";
        }
    }

    @address function bool vote(uint32 proposal_index, uint32 case_num) export {
        if(case_num == current_case && case_num > voted_case && proposal_index<proposals.length())
        {
            voted_case = case_num;
            votedWeights.set_length(proposals.length());
            votedWeights[proposal_index] += weight;
            return true;
        }
        return false;
    }

    @address function finalize() export {
        relay@global (){
            shardGather_reset();
            relay@shards (){
                __debug.print("Shard Vote: ", votedWeights);
                relay@global(auto shardVotes = votedWeights) {
                    for(uint32 i=0u; i<shardVotes.length(); i++)
                        proposals[i].totalVotedWeight += uint64(shardVotes[i]);

                    if(shardGather_gather()) {
                        last_result.case = current_case;
                        uint64 w = 0u64;
                        for(uint32 i=0u; i<proposals.length(); i++) {
                            if(proposals[i].totalVotedWeight > w)
                            {
                                last_result.topVoted = proposals[i].name;
                                w = proposals[i].totalVotedWeight;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Listing 4. The PREDA ballot contract

In the address scope function `init`, the controller of the contract is responsible for initiating the global scope variable `proposals`. This is done by issuing a relay transaction from the address scope to the global scope, which is corresponding to broadcast a relay transaction to the network in the underlying blockchain and to initialize the `proposals` and `last_result` variables. After the execution of the `init` function, the global scope variable `proposals` is initialized on all blockchain nodes.

In the address scope function `vote`, each voter can **directly** read the global scope variable `proposals` and also **directly** write the shard scope variable `voteWeights`, because the global scope variable is globally consistent, and the shard scope variable is consistent in each shard. At runtime, user-initiated transactions calling the `vote` function are executed by different shards based on address-to-shard mapping. Each transaction only changes the shard scope variable `voteWeights` in the corresponding shard.

In the address scope function `finalize`, the controller of the contract issues a relay transaction from the address scope to the global scope to reset the global scope variable `shardGatherRatio` and request all shards to report the value of the shard scope variable `voteWeights`. Each shard then sends a relay transaction with the value of `voteWeights` to the global. At the global scope on each blockchain node, the partial voting results `voteWeights` are accumulated to the final voting result `proposals`. After receiving the partial voting results from all shards, i.e., in the conditional sentence of calling the `shardGather_gather` function, the winner of the voting proposals is calculated and set as `last_result`.

2 MORE LANGUAGE FEATURES

PREDA language has enriched features. We introduce several of them in this section. Please refer to PREDA Language Specification [4] for more details.

2.1 Relay Statement with Lambda Expression

As shown in the previous example of the PREDA ballot contract, a relay statement can be defined as a lambda function. The format is quite similar to defining a function, except that:

- A function name is not needed. The compiler automatically generates a name for it.
- The scope of an anonymous function can be address (user-defined), shard, or global, based on the relay type.
- For each parameter, an argument must be provided as well.
- It is possible to use the "auto" keyword as the parameter type. In this case, the type is taken from the corresponding argument expression.

The relay function body is executed on the per-address context of target address, the per-shard context of the target shards, or the global context. It cannot be mixed with the current context scope on which the relay statement is invoked. There's two ways to specify a parameter in the relay lambda.

```
relay@someAddress (..., ^identifier, ...){
}
relay@someAddress (..., auto identifier = identifier, ...){
}
```

Listing 5. Relay Statement with Lambda Expression

They are equivalent but the first one is a simplified expression, where the ^ operator is used to capture variable by value, programmers can thus pass parameters to the function without renaming them.

2.2 System-reserved Functions

System-reserved functions are a group of special functions with the names reserved by PREDA for special purposes. They don't always have to be defined by a contract. But when they are, the definition must match a certain signature and will be invoked by the system at certain points. They are const functions and may emit relay transactions to do the actual task if needed. In addition, system-reserved functions may not access the transaction context (because they are not invoked by a transaction), or any part of the block context that is of payload or mined dependency (because they are executed before transactions in a block). There are two system-reserved functions in PREDA .

- **on_deploy():** it is a global function that is invoked when a contract is deployed. It can be used to do some initialization of the contract state. Its signature is: `function on_deploy()`.
- **on_scaleout():** it is a shard function that is invoked when the scaleout of the sharded blockchain happens, i.e. when the shard order of the blockchain system is increased by 1, i.e., from $n-1$ to n and the total number of shards doubles from 2^{n-1} to 2^n. On scaleout, each of the old 2^{n-1} shards is forked to two new shards: from $shard[i]$ to $shard[i]$ and $shard[i+2^{n-1}]$, where $0 <= i < 2^{n-1}$. The on_scaleout function is called 2^n times, once per shard. It can be used to split the old per-shard contract state into the two new shards. The signature is: `function on_scaleout(bool)`. The boolean parameter tells whether the current shard is forked in place (when false), or with offset 2^{n-1} (when true).

Note that although the system-reserved functions are defined in the PREDA language specification, they are not mandatory for underlying sharded blockchain systems. If a sharded blockchain doesn't support a system-reserved function, a PREDA contract using the function can still be compiled and executed on the sharded blockchain but doesn't have the functionality defined by the system-reserved function.

2.3 Multiple Contracts

In PREDA , a contract could interact with other contracts that are already deployed on the chain. To interact with another contract that contract must first be imported to the current contract as below:

```
import DAppName.ContractName [as AliasName];
```

`DAppName` and `ContractName` are the corresponding names assigned when deploying the imported contract. `AliasName` is an optional arbitrary identifier to reference it in the current contract. If `AliasName` is not given, `ContractName` will be used instead for referencing. The import must be declared before the new contract definition.

The PREDA language specification supports explicit import and implicit import. When a contract is imported by an import directive, it is explicitly imported. Besides that, a contract could also be implicitly imported if it is indirectly imported, like in the following example:

```
contract ContractA{
}

import MyDApp.ContractA as A;                          // ContractA is explicitly imported
contract ContractB{
}

import MyDApp.ContractB as B;                          // ContractA is implicitly imported
contract ContractC{
}
```

An implicitly-imported contract doesn't have a user-defined alias and can be referenced by its contract name by the compiler. In the above example, `MyDApp.ContractA` is referenced as `ContractA` in Contract C. To have a specific alias, it could be explicitly imported again. For example:

```
import MyDApp.ContractB as B;                          // ContractB is explicitly imported
import MyDApp.ContractA as A;                          // ContractA is explicitly imported as A
contract ContractC{
}
```

After importing a contract, all user-defined types and scopes from it could be accessed under the contract alias. As shown in the example below, after `ContractB` imports `ContractA`, the function `f` in `ContractB` has the scope `myScope` defined in `ContractA`.

```
contract ContractA{
  struct S{
    int32 i;
  }
  enum E{
    E0,
    E1
  }
  @myScope = global by address;
}
import MyDApp.ContractA as A;
contract ContractB{
  @A.myScope {
    A.S s;
    A.E e;
    function f(){
      s.i = 1i32;
      e = A.E.E0;
    }
  }
}
```

Similar to user-defined types, public functions defined in other contracts could also be directly referenced via the alias.

```
contract ContractA{
  struct S{
    int32 i;
  }
  enum E{
    E0,
    E1
  }
  @myScope = global by address;
  @myScope {
    function f(S s, E e) public{                            // only public functions can be called by other contracts
    }
  }
}
import MyDApp.ContractA as A;
contract ContractB{
  @A.myScope {
    A.S s;
    A.E e;
    function f(){
      A.f(s, e);                                            // call the public function f() from MyDApp.ContractA
    }
  }
}
```

The basic scope visibility rules hold for cross-contract calls, i.e., each scope can only call functions in the same scope, in the shard scope and const functions in the global scope.

2.4 Interfaces

Interfaces provide another way to work with multiple contracts. While only known contracts can be imported, interfaces enables interaction with arbitrary contracts that implements it. Interfaces are defined at the contract level. Each interface is a set of function definitions with empty bodies. Similar to regular functions, the functions of an interface must also reside in scopes:

```
contract A {
  interface Addable {
    @address {
      function Add(uint64 value);
    }
    @global {
      function uint64 GetTotal() const;
    }
  }
}
```

The above contract defines an interface `Addable` with two functions, each in a different scope. Interfaces can use scopes freely like scopes in contracts, including user-defined scopes and imported scopes from other contracts.

Contracts can choose to implement interfaces using the **implements** keyword at definition. A contract can choose to implement arbitrary number of interfaces, which can either be those defined in the same contract, or imported interfaces from other contracts.

```
import A;
contract B implements A.Addable, Printable {                // use "implements" to implement interfaces
  interface Printable {
    @global {
        function Print() const;
    }
  }

  @global {
     uint64 total;
     function uint64 GetTotal() public const {               // GetTotal() for A.Addable
       return total;
     }
     function Print() public const {                         // Print() for Printable
       __debug.print(globalTotal);
     }
  }
  @address {
     function Add(uint64 value) public {                     // Add() for A.Addable
       relay@global (^value) {
         total += value;
       }
     }
  }
}
```

The above contract implements two interface: `Printable` defined in the contract itself, and `Addable` defined in contract `A` from the previous section.

To implement an interface, a contract must implement all the functions defined in that interface, and the signature of the implemented function must match exactly the definition in the interface, i.e. the same function name, parameter list and type, return type, const-ness and scope. In addition, interface function must be implemented as **public**, since they are used for cross-contract calls. When a contract implements an interface, other contracts can interact with it via the interface. For example:

```
import B;                                                  // A is implicitly imported via B
contract C {
  @address {
    function test() {
      A.Addable addable = A.Addable(B.__id());             // initialize addable with contract B's id
      addable.Add(100u64);                                 // Calls B.Add() via the interface
    }
  }
}
```

In this example, a variable of interface type `A.Addable` is defined. Interface types can be initialized with a contract id. Here, it is initialized with a B's id using the build-in function `__id()` that is automatically generated for each contract. Once an interface variable is initialized, it can be used to call any function defined in the interface and routed to the corresponding implementation in contract B. With interfaces, a contract can interact with any other contract that implements the interface without knowing them. For example:

```
import A;
contract Adder {
  @address {
    function Add(A.Addable addable, uint64 value) public {
      addable.Add(value);
    }
  }
}
```

When the interface is defined, there is no need to import any other contract other than A. The function Add accepts an `A.Addable` interface as parameter, which could possibly be initialized by the id of any other contract that implements `A.Addable`. Note that if calling a function on an interface variable that is uninitialized, or initialized with the id of a contract that actually doesn't implement the interface, an error would occur and the contract execution will fail immediately.

3 DEPLOYMENT AND EXECUTION ENGINES

We have implemented a preview toolchain for the PREDA programming language. The PREDA language preview toolchain includes the pre-built PREDA compilers and execution engines, VSCode extensions, and sample smart contracts and execution scripts. Programmers can use VSCode to edit, compile, and execute PREDA smart contracts in a local blockchain simulator. The chain simulator uses multiple threads to mimic multiple shards of a sharded blockchain (one for each) on a single node. Programmers can also use the VSCode extension to check the contract execution results, e.g., address states on the chain. The supported command lines of the chain simulator and the script syntax of the chain simulator can be found in the PREDA toolchain user manual [5].

3.1 Contract State Management in Chain Simulator

In the implementation of the PREDA chain simulator, we use the Map of the contract ID to the contract states as the basic data structure. The basic chain state is divided into pieces represented as `<scope, [shard/address], contract map>`. The chain simulator uses this structure to track partial changes to the entire chain state. At runtime, the execution of transactions, including those in chain blocks and orphaned blocks, results in newly modified contract states. We store incremental changes to contract states as follows:

```
<Global State>   ::= <Contract Map>
<Shard State>    ::= <Contract Map>
<Address State>  ::= MAP(<Address> => <Contract Map>)
<Contract Map>   ::= MAP(<Contract Id> => <State History>)
<State History>  ::= ARRAY(<State on Chain>)
```

```
<State on Chain> ::= <Height, Block Id, Opaque State Data>
```

Note that the array `<State History>` is sorted as the chain height increases. In the chain simulator, the total chain states include: (1) the base chain states for archived blocks that are not reverted, and (2) a history of incremental changes made by all blocks since the height of the last archived block, including blocks in forked chains.

3.2 Contract Compilation

We employ a two-stage process to compile smart contracts written in PREDA to native code. The first stage is a transpiler that interprets the PREDA source code and converts it to an intermediate representation, which is then consumed by the second stage to generate the binary code. We use C++ source code as our intermediate representation.

Our transpiler uses ANTLR [1] to generate the parser code based on the PREDA language specification. Then, the transpiler walks through the abstract syntax tree (AST) to extract state variables, user-types, and function definitions, and outputs the corresponding C++ code. The output of the transpiler for each contract is a single C++ complication unit. To compile C++ code to binary code, we use MinGW-w64 [2] on Windows and g++ on Linux. Their output are dynamic libraries that can be loaded on demand at runtime when a contract needs to be executed on the chain simulator.

Other than the local chain simulator, we are developing an experimental sharded blockchain for PREDA , with the supports of blockchain wallets and explorer services. We are also integrating EVMone into the chain simulator to implement the interoperability between Solidity and PREDA.

REFERENCES

[1] [n. d.]. ANother Tool for Language Recognition. https://www.antlr.org/.
[2] [n. d.]. MinGW-w64. https://www.mingw-w64.org/.
[3] EthereumDev. 2020. Transfers and Approval of ERC-20 Tokens from a Solidity Smart Contract. https://ethereum.org/pt-br/developers/tutorials/transfers-and-approval-of-erc-20-tokens-/\from-a-solidity-smart-contract/.
[4] PREDA Dev Team. 2022. PREDA Language Specification.
[5] PREDA Dev Team. 2022. PREDA Toolchain User Manual.