Appendix A: Introduction to the Model-Xchain Method

This appendix provides an detailed description of the Model-Xchain method, which follows a three-layer process for model-driven cross-chain business development, as illustrated in Figure 1.

The methodology begins at the *Modeling Layer*, where cross-chain operations and dependencies are identified and formalized into a domain-specific script. At the *Transformation Layer*, the script is parsed into a Directed Acyclic Graph (DAG) model and undergoes structural and semantic validation to ensure model correctness. Finally, the *Execution Layer* deploys the validated DAG to multiple blockchains, enabling coordinated and atomic execution of cross-chain operations. This layered design bridges business-level specification with executable coordination logic, ensuring both development automation and runtime reliability.

This layered design bridges business-level specification with executable coordination logic, ensuring both development automation and runtime reliability.

Detailed explanations of each step and the associated methodologies will be provided in the subsequent sections.

1 LAYER 1 — MODELING LAYER: SCRIPT-BASED CROSS-CHAIN BUSINESS MODELING

As a first step, a script-based method is designed as a DSL to describe the cross-chain business. This section describes 1) the identification of events and dependencies within cross-chain operations *(Step 1.1)*, and 2) the construction of a corresponding script that reflects these cross-chain operations *(Step 1.2)*.

1.1 Identifying Cross-Chain Operations

This step focuses on recognizing key events and their interdependencies within cross-chain operations, as shown in Figure 1. It includes: event definition *(Step 1.1.1)* and dependency definition *(Step 1.1.2)*.

To ensure independence from specific blockchain platforms, we extract the commonalities among different chains to eliminate the differences in generating and executing the transaction graph. Specifically, blockchain platforms are essentially decentralized distributed key-value databases [7, 10]. Therefore, we can extract their capabilities for data reading and writing operations. By integrating an adapter with smart contracts, we can encapsulate the data reading and writing operations on different chains. In addition, we can shield other blockchain details, such as consensus mechanisms and storage structures. It is worth noting that, due to significant differences in smart contracts of different blockchains, we also abstract the contract layer and provide scripts to enable uniform descriptions for developing cross-chain business applications. During the development of cross-chain businesses, adapters are used to interface with the underlying API of each blockchain to ensure the compatibility of our approach with different blockchains. Further script details will be discussed in Section 1.2.

Regarding **event definition (Step 1.1.1)**, cross-chain business processes involve multiple operations such as read, write, and computation. We define six types of events to represent key operations in cross-chain business processes. These events are considered graph nodes. Therefore, we need to summarize the basic blockchain operations or capabilities to identify the type of graph node.

In cross-chain interaction, not only data reading and writing are involved, but also operations such as logic execution and data calculation based on the smart contract of each chain. These operations can be abstracted into the abilities of data processing. Consequently, we summarize three common abilities: data reading, data writing, and expression computation, which are the nodes (events) in the dependency graph. Furthermore, in realistic scenarios, some parameters can be input externally [12], and different conditions can determine the next step of behavior or calculation method [8], and ultimately transactions can be canceled under certain conditions [4]. Therefore, this paper has added three abilities, namely parameter input, condition, and rollback, to describe different events in cross-chain commits.

In conclusion, based on the abilities, the dependency graph defines the following nodes: Input node, reading node, writing node, declaration node, condition node, and rollback node. The corresponding relationships between events and nodes and their meanings are summarized in Table 1. After defining all node types, the next step is to discuss the specific information fields included in each node.

Table 1: Graph Node Definition

Node Name	Description
Input Node	Operations need an external parameter
Reading Node	Read the data of a key from a chain
Writing Node	Write key-value pairs to a chain
Declaration Node	Compute the expression and generate a new variable
Condition Node	Execute certain operations when a condition is met
Rollback Node	Declare the failure of operation and revoke the previous write operation

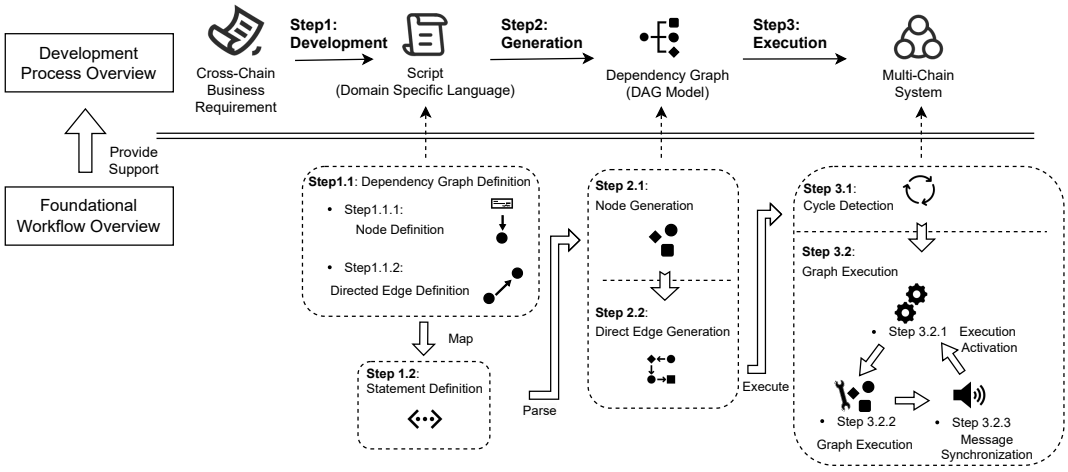

Figure 1: Overview of Model-Xchain Method

Each node contains its own field information and is responsible for a different event. The input node represents the parameters passed from the external source for cross-chain transactions. The reading node represents the reading of data from a specific chain. The writing node represents the data to be written to a specific chain. Values can be expressions, reducing the number of variable declaration nodes in the graph. The declaration node is used to compute and generate a new variable from an expression or overwrite the value of an existing variable. The condition node represents different events that need to be processed under different conditions. The conditions can be used to describe branching structures, and nested branching structures can be implemented for complex conditional scenarios. A condition node contains multiple child nodes, including other condition nodes, forming a recursive structure. These child nodes can also be organized using dependency relationships. The conditional node stores the condition expression, and maintains separate lists to execute when the condition is met or not. The rollback node cancels the current cross-chain transaction and reverts the write operations. The rollback overwrites the newly written values with the original values on the chain to ensure data operation atomicity. It is typically a child node of a condition node and does not require additional fields. The original state is recorded by the reading node during data retrieval. Rollback operations are completed by initiating a series of reversal transactions.

Regarding event **dependency definition (Step 1.1.2)**, the dependencies are identified to establish the relationships between events, ensuring the correct sequencing and execution of operations across multiple chains in the execution process. For instance, the data to be written into a chain in a write event may depend on the result of a previous read or computation event. To capture these dependencies between events, we introduced directed edges.

Graphic nodes can be connected through node fields, as shown in Table 2. To identify directed edges, we list the variables each type of node can generate and the fields on which they may depend. The name input from the outside is represented by "name", the newly generated variable name is represented by "var_name". Since blockchain storage is based on a key-value database, it also involves "key" and "value". "expression" represents a computable expression, and "condition" refers to conditional statements that return true or false. Moreover, due to the nested nature of conditional nodes, they involve variable generation and dependency analysis within their subnodes. The "true_sub_node_list" and "false_sub_node_list" fields can be used to examine variable generation and dependency analysis within the child nodes of the condition node. The integrated result can represent the entire situation of this conditional node.

Table 2: Graph Directed Edges Identification

Node Type	Generation of Variable Fields	Dependency of Variable Fields
Input Node	name	-
Reading Node	var_name	key
Writing Node	-	key, value
Declaration Node	var_name	expression
Condition Node	true_sub_node_list, false_sub_node_list	condition, true_sub_node_list, false_sub_node_list
Rollback Node	-	-

The directed edges, or connections between nodes in the graph, are based on the values of their respective fields. Directed edges can represent the dependencies with the generator node pointing to the dependent node. This relationship implies a sequential execution order, with the tail node preceding the head node.

In summary, directed edges always point from nodes that generate variables (Table 2, second column) to nodes that depend on other variables (Table 2, third column). Figure 2 shows an example of inventory statistics. The input nodes with IDs "1" and "2", respectively, read in two numbers, identified by the variable names "Inventory_A" and "Inventory_B". The declaration node with ID "3" needs to sum them up and store the sum in the variable "Total_inventory". Therefore, two directed edges originate from nodes "1" and "2" and point towards node "3".

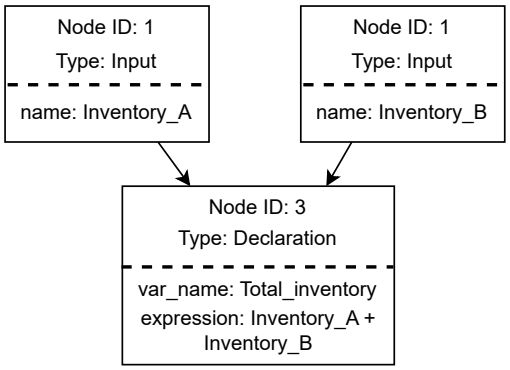

Figure 2: Example of Directed Edges Generation

1.2 Script Construction

Following the identification of events and dependencies, this step focuses on the creation of a script that reflects the defined cross-chain operations. The script is generated to capture the events and dependencies for executing these operations.

A script consists of multiple statements, and the definition of statements here is based on the node types and the node fields, to facilitate the conversion of the script into a dependency graph. In addition, to align with the developers' coding practices, script statements are defined based on Shell script syntax and Backus-Naur Form (BNF), as shown in Listing 1.

BNF is a notation used to describe the syntax of programming languages with formal grammars [5, 9]. BNF's production rules generate valid sentences or program statements that comply with the defined grammar, which has been widely adopted in programming language design [11]. The following is a detailed description of the cross-chain script statement definition.

Table 3: Script Statement Definition

Node types	Statement Expression
Input Node	loadParam(<var_name>)
Reading Node	<var_name>=read(<chain_id>, <key>)
Writing Node	write(<chain_id>, <key>, <value>)
Declaration Node	<var_name>=<expression>
Condition Node	if <conditions >then {<true_sub_node_list >} [else {<false_sub_node_list >}] fi
Rollback Node	-

The variable name is represented by "<var_name>", and the unique identifier for the blockchain is represented by "<chain_id>". Since blockchain storage is based on a key-value database, it also involves "<key>" and "<value>". "<expression>" represents a computable expression, "<conditions>" refers to conditional statements that return true or false, and "<statement>" is a general term for statements.

As shown in Table 3, according to the corresponding node types, the statements used in a business context are listed below:

- The loadParam statement indicates cross-chain business' dependency on external input parameters.
- The read statement indicates the need to read specific data from a certain chain and assign them to a variable.
- The write statement indicates the need to write a result to a specific chain.
- The declaration statement indicates the calculation of an expression and generates a new value.
- The condition statement handles branching in the cross-chain business, where different operations are executed based on different conditions. Similarly to shell scripts, the "else" part can be omitted.
- The rollback statement is used to reverse cross-chain commitments.

Figure 3 shows an example script for a cross-chain business case. In this case, the fishery, the logistic provider and the shop retailer operate on different blockchains (i.e. fishery_chain, express_chain and shop_chain respectively in the script). Fisheries sell batches of seafood to shop retailers via transport services provided by logistics providers.

Within the script, as shown in Figure 3, in addition to the statements and grammar aforementioned, "$" means variable, "+" means string splicing. The script is enclosed by an atomic block (line 1), which defines an atomic cross-chain execution

Listing 1: Script Definition in BNF

```
1                <Script>::= <Statement> {<Statement>}
2             <Statement>::= <LoadParamStatement> | <DeclarationStatement> |
3                            <WriteStatement> | <ConditionalStatement> |
4                            <ReadStatement> |  <RollbackStatement>
5    <LoadParamStatement>::= loadParam(<VariableName>)
6         <ReadStatement>::= <VariableName> = read(<ChainId>, <StringExpression>)
7   <DeclarationStatement>::= <VariableName> = <Expression>
8  <ConditionalStatement>::= if <ConditionalExpression> then \n {<Statement>\n}
9                            [else \n { <Statement> \n} \n] fi
10       <WriteStatement>::= write(<ChainId>, <StringExpression>, <Expression>)
11     <RollbackStatement>::= rollback
```

scope. All operations within this block are treated as a single atomic unit: if any exception occurs during execution, the runtime automatically triggers a cascade rollback to revert all previously executed write operations associated with this block. We use "loadParam()" function (line 2) as the input statement to retrieve the seafood ID; The “read()“ function (line 3) then queries the fish weight from FISHERY_CHAIN based on the given ID. Two conditional branches (lines 4–10) are used to determine the corresponding transport_price and transport_time according to the weight range. If the weight value is illegal, an exception is thrown (line 11), which causes the execution to enter an abnormal state and triggers rollback at runtime. The subsequent “write()“ functions (lines 13–14) record the transport price and transport time to EXPRESS_CHAIN. The “now()“ function (line 15) retrieves the current system time, and the final “write()“ function (line 16) stores the expected arrival date to SHOP_CHAIN. The exact values of the variables will be confirmed during the execution of the DAG.

```
1.  atomic {
2.    loadParam(fish_id)
3.    weight = read(FISHERY_CHAIN, weight_ + $fish_id)
4.    if $weight >= 0 and $weight <= 15 then
5.      transport_price = 30
6.      transport_time = 2
7.    else if $weight <= 40 then
8.      transport_price = 40
9.      transport_time = 3
10.   else
11.     throw Exception("Invalid fish weight")
12.   fi
13. write(EXPRESS_CHAIN, transport_price: $fish_id, $transport_price)
14. write(EXPRESS_CHAIN, transport_time: $fish_id, $transport_time)
15. expected_date = now() + $transport_time
16. write(SHOP_CHAIN, expected_date: $fish_id, $expected_date)
17. }
```

Figure 3: Sample Script of Seafood Supply Chain

2 LAYER 2 — TRANSFORMATION LAYER: AUTOMATIC DAG GENERATION

In Step 2, the script must be automatically parsed and converted into the corresponding dependency graph for further execution. Therefore, this section describes 1) script parsing for node generation *(Step 2.1)* and 2) dependency analysis to generate dependency graph *(Step 2.2)*.

2.1 Nodes Generation from Event

Through script parsing, we can obtain information about the nodes and their respective fields for graph node generation *(Step 2.1)*.

We introduce the concept of Recursive Descent Parsing and LL(1) for parsing. LL(1) is a class of context-free grammars which stands for "Left-to-right, Leftmost derivation with one symbol lookahead [3]." And Recursive Descent Parsing is a top-down parsing technique used to analyze programming language syntax or other formal language [1]. This method is intuitive and straightforward to implement and is widely used in practice [6].

Since the script statements grammar does not contain left recursion and follows the LL(1) grammar, the recursive descent parsing method can be used for analysis. Due to the recursive nature of the analysis process, the contexts are different. This requires global data structures to store node IDs, node fields, and node generation status. As shown in Algorithm 2, the "generate_vars" and "dependent_vars" structures record variable-node mappings in an inverted index format, for subsequent node connections.

The process of script parsing is to call the statement parsing function parse_one_sentence to process each line of the script, as shown in Figure 4. Its main purpose is to determine the type of the current line and invoke the corresponding parsing subfunction for node generation.

Figure 4 illustrates the statement parsing process that generates the graph nodes. The core process involves three steps: 1) splitting the sentence into a word list containing multiple words, 2) determining the type of the current line based on the keywords, and 3) invoking the corresponding parsing sub-function for node generation. The six subfunctions depicted in the state transition diagram map the six types of script statements mentioned before. After completing the parsing of a line, graph nodes and relevant variables

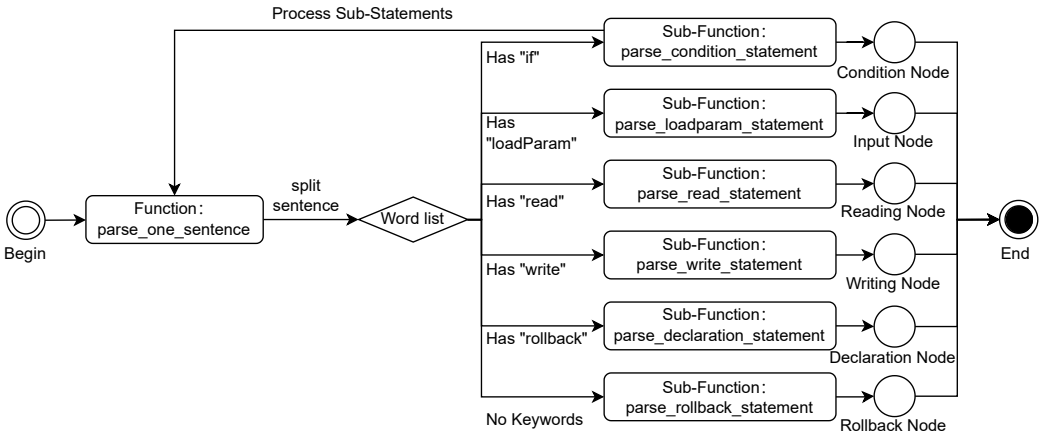

Figure 4: Script Parsing Process for Dependency Graph Generation

are extracted. As shown in algorithm 1, by parsing each line, a parse_list is obtained, which contains the elements in parsed_node, parsed_gen_vars, and parsed_dep_vars. These elements are accumulated and recorded in generate_vars and dependent_vars. In other words, the main algorithm generates and records all the nodes and their dependent fields based on the graph. This information can be further utilized to construct a complete cross-chain event dependency graph.

Algorithm 1 Main Script Parsing Algorithm

Input: sentences of lines in script

```
for line in lines do
    parsed_list ← parse_one_sentence(line)
    for parsed_part in parsed_list do
        parsed_node, parsed_gen_vars, parsed_dep_vars ← parsed_part
        for parsed_gen_var in parsed_gen_vars do
            generate_vars[parsed_gen_var] ← parsed_node
        end for
        for parsed_dep_var in parsed_dep_vars do
            dependent_vars[parsed_dep_var] ← parsed_node
        end for
    end for
end for
```

2.2 Directed Edges Generation with Dependency Analysis

After generating graph nodes, we need to establish directed edges to generate a complete dependency graph *(Step 2.2)*.

Sequential execution of each node by its ID can result in inefficiency. Therefore, our approach parallelizes mutually independent nodes in order to improve the graph's execution efficiency within each chain. For instance, in a blockchain, writing nodes require consensus and must be executed sequentially, whereas reading nodes can be executed simultaneously.

As shown in Algorithm 2, the node connections are based on the field information within each node. Among these, nodes with variables on which the current node depends need to be connected in ascending order of their node_id. This is because node_id reflects the dependencies and execution order among nodes. Different execution sequences can lead to different results, thus causing inconsistencies. As for the nodes that depend on the generated result of the current node, there is no direct order among them. This is because there is no dependency between the nodes and the results of node execution do not interfere with each other. In summary, it is only necessary to ensure the relevant operations are completed before the variable changes.

We can illustrate this with a simple example. As depicted on the top side in Figure 5, nodes "1", "2", and "3" all depend on variable "i", thus requiring execution in ascending order based on their node IDs. On the bottom side, nodes "4" and "8" both depend on the variable "i". However, nodes "5", "6", and "7" only depend on the variable "i" from node "4" and have no inter-dependencies among themselves. Therefore, nodes "5", "6", and "7" do not have a specific execution order requirement. Furthermore, node "8" is executed at the end.

Through dependency analysis, all nodes can be connected based on the relationships of variable generation and dependency. Finally, it is only necessary to return the starting node (entry point) of the event dependency graph, which is a variable generation node with an in-degree of zero.

Continuing to use the aforementioned cross-border seafood trading scenario as an example, it involves the fishery, the

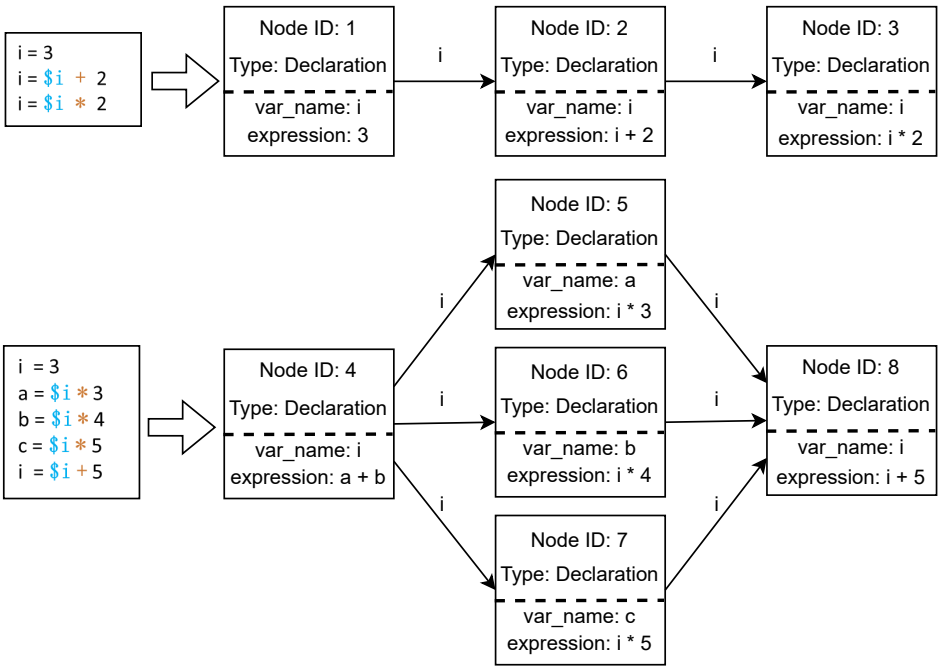

Figure 5: Examples of Node Dependency Generation

Algorithm 2 Graph Node Connection Algorithm

Input: generate_vars, dependent_vars
Output: start list of a graph starts

```
variables ← dependent_vars.keys()∩ generate_vars.keys()
starts ← [ ]
for variable in variables do
    gen_vars ← sort(generate_vars[variable], node_id, ascending)
    dep_vars ← sort(dependent_vars[variable], node_id, ascending)
    starts.append(gen_vars[0])
    j ← 0
    for i=1 to len(gen_vars) do
        gen_vars[i].parents.append(gen_vars[i − 1])
        repeat
            dep_vars[j].parents.append(gen_vars[i − 1])
            gen_vars[i].parents.append(dep_vars[j])
            j ← j + 1
        until dep_vars[j].node_id > gen_vars[i].node_id
    end for
    repeat
        dep_vars[j].parents.append(gen_vars[len(gen_vars) − 1])
        j ← j + 1
    until j >= len(dep_vars)
    return starts
end for
```

logistic provider, and the shop retailer. The fishery needs to sell seafood to the shop retailer through the logistic provider. As shown in Figure 6, the dependency graph illustrates the interactions and dependencies between the fishery_chain, express_chain and shop_chain, including reading and writing data, conditional judgments, and rollbacks. After entering the ID in node 1, node 2 reads the seafood product weight. In case the weight is illegal, node 9 triggers a rollback operation. If not, declare the variable values at nodes 4,5,7,8 based on different criteria. Finally, nodes 10 and 11 perform chain writing operations.

Continuing to use the aforementioned cross-border seafood trading scenario as an example, it involves the fishery, the logistic provider, and the shop retailer. The fishery needs to sell seafood to the shop retailer through the logistic provider. As shown in Figure 6, the dependency graph illustrates part of cross-chain process, including the interactions and dependencies between the fishery_chain and express_chain. After entering the ID at node 1, node 2 reads the weight of seafood from fishery_chain. Nodes 3 and 6 make judgments respectively according to different criteria. If the weight is not legal then node 9 is triggered to perform a rollback operation, which aborts the entire dependency graph. Otherwise, different procedures are executed according to the read weight value: if the weight is between 0 and 15 kilograms, declare the transportation price as 30 dollars (node 4) and the transportation time as 2 days (node 5). If the weight is more than 15 kilograms but less than 40 kilograms, the transportation

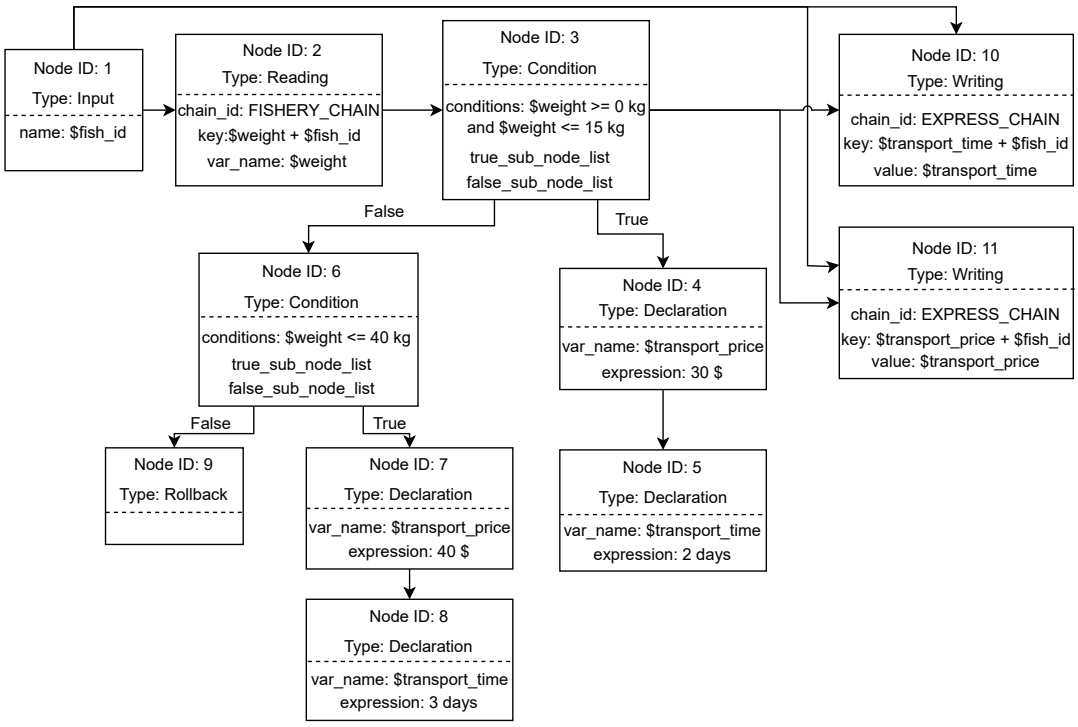

Figure 6: Sample Dependency Graph of Seafood Logistics Supply Chain (Part)

price is declared to be 40 dollars (node 7) and the transportation time is 3 days (node 8). Finally, nodes 10 and 11 write the obtained transportation price and time to express_chain respectively.

In conclusion, due to the correspondence between the script statements and the nodes, the graph's edges analysis is also straightforward. Therefore, we can effectively generate dependency graphs for various operations within cross-chain business scenarios using scripts.

2.3 Static Validation and Validity Checks (Step 2.3)

Before deployment, each generated DAG model in ModelXchain undergoes a static validation process to ensure its structural integrity, logical soundness, and rollback completeness. The validation engine analyzes the DAG $G = (V, E)$ produced by the parser, where V represents operation nodes and E denotes directed dependencies among them. Three complementary checks are performed: (1) *Cycle Detection*, (2) *Dangling-Dependency Verification*, and (3) *Compensation Completeness*. These procedures identify structural defects and invalid references at the modeling level, preventing runtime failures during cross-chain execution.

(a) Cycle Detection. A Directed Acyclic Graph must represent one-way execution dependencies. Any cycle would imply mutual waiting or non-terminating rollback propagation. In DAGs, a cycle refers to a path that starts at node A, traverses several edges, and eventually returns to node A. The existence of cycles can potentially lead to circular dependencies among transaction events, resulting in deadlocks in cross-chain transactions. By performing cycle detection, we can determine a topological execution sequence that guarantees the completion of the business process within a finite number of steps, ensuring consistency and reliability.

Hence, detecting and handling cycles in the event dependency graph is a key step prior to cross-chain event execution. Algorithm 3 shows the cycle detection algorithm.

The algorithm utilizes a topological sorting of the dependency graph to determine the presence of cycles. At each step, nodes with an in-degree of 0 are removed from the graph. If there are remaining nodes in the graph but no nodes with an in-degree of 0 can be found, it indicates the presence of cycles. Conversely, if all nodes are successfully removed from the graph, it indicates the absence of cycles.

(b) Dangling-Dependency Check. Each dependency edge must connect two valid nodes defined within the same model. Errors in script references or missing imports may produce "dangling" dependencies that compromise the execution order.

Algorithm 3 Cycle Detection Algorithm

Input: start list of a graph starts
Output: if graph contains a loop

```
q ← Queue()
for start in starts do
    q.enqueue(start)
end for
repeat
    loop ← true
    q.enqueue(null)
    repeat
        cur ← q.dequeue()
        if cur = null then
            break
        end if
        if cur.parents.empty() then
            loop ← false
            for child in cur.children do
                child.parents.remove(cur)
            end for
        end if
        for child in cur.children do
            q.enqueue(child)
        end for
    until q.empty()
    if loop = true then
        return true
    end if
until q.empty()
return false
```

Algorithm 4 Dangling-Dependency Verification

Input: Node set V, edge set E
Output: Report missing source or target nodes

```
valid_ids ← {v.id | v ∈ V}
for (u, v) ∈ E do
    if u.id ∉ valid_ids then
        report(DanglingSource, u)
    end if
    if v.id ∉ valid_ids then
        report(DanglingTarget, v)
    end if
end for
```

This verification ensures that every edge (u, v) connects existing nodes. When a node reference is missing (for instance, a variable produced in one sub-script but never imported), the validator provides diagnostic feedback pinpointing the faulty line and variable name. This check enforces referential integrity between event nodes and ensures that dependency propagation is well-defined.

(c) Rollback Completeness. To guarantee reversibility, each write operation in the DAG must declare a corresponding compensation or rollback node. This rule ensures that every side effect in the business process can be undone in case of partial execution failures.

Algorithm 5 Rollback Completeness Check

Input: Node set V with attributes `type`, `has_comp`
Output: Report missing rollback for write nodes

```
for u ∈ V do
    if u.type = WRITE ∧ ¬u.has_comp then
        report(MissingRollback, u)
    end if
end for
```

The validator checks whether each `WRITE` node has an associated compensation declaration or rollback mapping. When missing, the model is flagged as invalid since it violates the atomicity assumption—operations without compensation would leave irreversible states on-chain. This constraint enforces the cascading rollback semantics that underpin ModelXchain's atomic cross-chain commit protocol.

3 LAYER 3 — EXECUTION LAYER: DAG-ORIENTED ATOMIC CROSS-CHAIN EXECUTION

To ensure atomicity and consistency of the data operations, following the generation of the dependency graph, the dependency graph must be executed to complete the cross-chain business process in Step 3. Before the execution, cycle detection *(Step 3.1)* is required. The dependency graph will then be executed in a multichain system *(Step 3.2)*. The following section provides a more detailed description of these processes.

As shown in Algorithm 3, by performing cycle detection *(Step 3.1)*, we can determine a topology execution sequence to ensure that the business process is completed within a limited number of steps, ensuring consistency and reliability. In the dependency graph, a loop refers to the path that starts at node N, passes through several edges, and ultimately returns to node N. The existence of loops may cause circular dependencies between transaction events, leading to deadlocks in cross-chain transactions.

Therefore, detecting and processing the cycles in the event dependency graph is a critical step before executing cross-chain events. We use the topological sorting method [2] to determine whether there are cycles in the graph. If there is a topological order in the graph, it indicates that there are

no cycles. Only acyclic graphs can be input into subsequent execution steps. A graph that contains cycles will be rejected for execution, which eliminates the possibility of human error in developing scripts and ensures their consistency and reliability.

3.1 DAG Deployment on Blockchains

The execution of the DAG model in Model-Xchain is handled by adapters deployed on the nodes of different blockchains. The adapters are specifically designed for each type of blockchain technology to realize the invocation of the corresponding smart contract API. Each adapter is responsible for executing a cross-chain transaction according to the received DAG that is identical across the nodes. These adapters invoke smart contract APIs to perform the operations required by each DAG node.

Given the dependencies between blockchains in cross-chain transactions, the adapters on different blockchains will broadcast information to each other, sharing the execution status and data of the DAG. This coordination among adapters could be performed by a decentralized coordinator, allowing for synchronized execution across multiple blockchains without relying on a centralized coordinator.

When a rollback occurs, compensatory transactions are initiated by adapters through smart contract APIs. When an adapter encounters failures, it broadcasts a failure message to all other adapters. This triggers rollback transactions on other blockchains, leading to a cascading rollback up to the DAG's entry point. An absence of an adapter can also result in failure messages being broadcast by other adapters. Therefore, the failure tolerance can be ensured.

3.2 Execution of Dependency Graph

Business process execution across multiple blockchains should be automated while ensuring atomicity. As a result, the graph execution algorithm is designed for collaborative graph execution after cycle detection *(Step 3.2)*.

In step 3.2, shown in Figure 1, the dependency graphs are executed through a loop consisting of three main steps.

As shown in the algorithm 6, before cross-chain operations, the input parameters need to be added to the variable pool (var_pool). The working_member is a list that keeps track of the identifiers of all participating chains. The loop continues until all chains in the working_member list have completed their execution. Executing_queue records all the graph nodes that need to be executed. The entry point for execution is the nodes with an in-degree of zero. The listener is then launched as a daemon program to continuously monitor and record incoming messages from other chains.

Then the main loop begins for dependency graph execution, which can be divided into three steps. To begin with, the message processing function (handle_message function) is invoked in order to process messages from other chains *(Step 3.2.1)*. This step is skipped in the initial iteration. Then, the execution function (execute_once function) executes the executable nodes in the graph and generates results and messages *(Step 3.2.2)*. Finally, the generated messages are broadcast to other chains *(Step 3.2.3)*. Following is a description of each step:

Algorithm 6 Main Loop of Graph Execution

Input: The key-value pairs **param_pairs** passed when invoking the graph, the list of all chain IDs participating in cross-chain operations is referred to as **chain_ids**, start list of a graph **starts**

```
for key, value in param_pairs do
    var_pool[key] ← value
end for
working_member.add_all(chain_ids)
executing_queue.enqueue_all(starts)
start listening message from other chain
repeat
    messages ← messages broadcast from other chains
    handle_message(messages)
    results ← execute_once()
    broadcast results to other chains
until working_member.empty()
```

Algorithm 7 Message Handling Process

Input: current chain's id **self_chain_id**

```
repeat
    result ← received_queue
    if result is finished message then
        working_member.remove(result.chain_id)
    end if
    if result is rollback message then
        revoke previously completed write operations
        working_member.remove(result.chain_id)
        working_member.remove(self_chain_id)
        send rollback message to other chains
    end if
    executed_nodes.add_all(result.node_ids)
    for key, value in result.kv_pairs do
        var_pool[key] ← value
    end for
until received_queue.empty()
```

In the *execution activation phase (Step 3.2.1)*, the DAG nodes (operations) that have no unfinished dependencies are identified and activated. These operations are then translated into blockchain-specific calls. If have, messages broadcasted by other chains are received and processed as shown in Algorithm 7. These messages contain the execution state and the newly generated variables on the respective chains.

Following is the message types and handling approaches: 1) Completion message: Indicates that all nodes of a chain have completed their cross-chain execution. After receiving a completion message, we will remove the corresponding chain ID from the working_member list. This means that all cross-chain data operations on this blockchain have been completed. 2) Rollback message: Indicates a node execution failure that requires the revocation of write operations. After receiving the rollback message, we will initiate the rollback transaction and revert all data operations to the state before the dependency graph was executed to ensure atomicity and consistency. 3) Data message: Indicates normal execution of nodes in the current round and broadcasts node fields for the next round of node execution. After receiving a data message, we will mark the corresponding node in the graph as executed and add the variables from the message to the variable pool for future use. Hence, other graph nodes relying on this node are able to continue executing in the next round.

In the *graph node execution phase (Step 3.2.2)*, all nodes are executed in the dependency graph with an in-degree of zero. This execution process generates new states and variables that need to be notified to other chains. As shown in Algorithm 8, this phase has multiple rounds, and each round aims to run nodes with an in-depth of zero in the graph and generate messages for further broadcasting. The nodes are executed sequentially from the executing_queue. If the queue is empty, indicating the completion of tasks in the current chain, a completion message is returned. Otherwise, the nodes in the executing_queue are processed in order. If an error occurs during execution, a rollback message is returned. If all nodes are executed successfully, the nodes that cannot be executed are added back to the queue for the next round. The newly generated variables and node IDs from this round are packaged in a data message and returned.

In *message synchronization phase (Step 3.2.3)*, blockchains broadcast their states, variables, and other relevant information to other chains, achieving multi-chain collaboration execution with the dependency graph. Broadcasting different types of messages triggers relevant actions on other chains, thereby ensuring the consistency of cross-chain business execution.

Algorithm 8 Graph Node Execution Algorithm

Input: current chain's id **self_chain_id**
Output: message need to broadcast to other chain

```
if executing_queue.empty() then
    working_member.remove(self_chain_id)
    return finishedmessage
end if
batch_new_vars ← dict()
batch_broadcasts_node_ids ← list()
batch_unfinished_nodes ← list()
repeat
    node ← executing_queue.dequeue()
    new_vars, back_vars, broadcast_node_ids,
unfinished_nodes ← execute_node(node)
    if new_var = null then
        revoke previously completed write operations
        working_member.remove(self_chain_id)
        send rollback message to other chains
    end if
    for k, v in new_vars do
        var_pool[k] ← v
        batch_new_vars[k] ← v
    end for
    for k, v in back_vars do
        if back_pool[k] = null then
            back_pool[k] ← v
        end if
    end for
    batch_broadcast_node_id.add_all(node_ids)
    batch_unfinished_node.add_all(unfinished_nodes)
    if node not in unfinished_nodes then
        for child in node.children do
            child.parents.remove(node)
            if len(child.parents) = 0 then
                executing_queue.enqueue(child)
            end if
        end for
    end if
until executing_queue.empty()
executing_queue.enqueue_all(batch_unfinished_nodes)
create data message result
result.node_ids ← batch_broadcasts_node_ids
result.kv_pairs ← batch_new_vars
return result
```

REFERENCES

[1] Mikhail Barash. 2013. Recursive descent parsing for grammars with contexts. In *SOFSEM 2013 student research forum Špindleruv Mlỳn, Czech Republic, 26-31 January.* 10–21.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms.* The MIT Press, Cambridge, MA.

[3] Donald E Knuth. 1965. On the translation of languages from left to right. *Information and control* 8, 6 (1965), 607–639.

[4] Léonard Lys, Arthur Micoulet, and Maria Potop-Butucaru. 2020. Atomic cross chain swaps via relays and adapters. In *CryBlock@MOBICOM 2020: Proceedings of the 3rd Workshop on Cryptocurrencies and Blockchains for Distributed Systems, London, UK, September 25, 2020.* ACM, Shenyang, Liaoning, China, 59–64. https://doi.org/10.1145/3410699.3413799

[5] Daniel D McCracken and Edwin D Reilly. 2003. Backus-naur form (bnf). In *Encyclopedia of Computer Science*. John Wiley and Sons Ltd., GBR, 129–131.

[6] Alexander Okhotin. 2007. Recursive descent parsing for Boolean grammars. *Acta Informatica* 44, 3-4 (2007), 167–189. https://doi.org/10.1007/s00236-007-0045-0

[7] Babu Pillai, Kamanashis Biswas, and Vallipuram Muthukkumarasamy. 2020. Cross-chain interoperability among blockchain-based systems using transactions. *Knowl. Eng. Rev.* 35 (2020), e23. https://doi.org/10.1017/S0269888920000314

[8] Hong Su, Bing Guo, Jun Yu Lu, and Xinhua Suo. 2022. Cross-chain exchange by transaction dependence with conditional transaction method. *Soft Comput.* 26, 3 (2022), 961–976. https://doi.org/10.1007/s00500-021-06577-5

[9] Gordon Whitney. 1969. An extended BNF for specifying the syntax of declarations. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1969 Spring Joint Computer Conference, Boston, MA, USA, May 14-16, 1969 (AFIPS Conference Proceedings, Vol. 34)*, Harrison W. Fuller (Ed.). AFIPS Press, New York, NY, USA, 801–812. https://doi.org/10.1145/1476793.1476929

[10] Canghai Wu, Jie Xiong, Huanliang Xiong, Yingding Zhao, and Wenlong Yi. 2022. A Review on Recent Progress of Smart Contract in Blockchain. *IEEE Access* 10 (2022), 50839–50863. https://doi.org/10.1109/ACCESS.2022.3174052

[11] Lieh-Ming Wu, Kuochen Wang, and Chuang-Yi Chiu. 2004. A BNF-based automatic test program generator for compatible microprocessor verification. *ACM Trans. Design Autom. Electr. Syst.* 9, 1 (2004), 105–132. https://doi.org/10.1145/966137.966142

[12] Zheng Xu, Chaofan Liu, Sitong Wang, Peng Zhang, Tun Lu, and Ning Gu. 2022. An AST-based consistency maintenance scheme for cross-chain digital assets. *CCF Trans. Pervasive Comput. Interact.* 4, 2 (2022), 142–157. https://doi.org/10.1007/s42486-022-00096-4