

Article

Research on Smart Contract Verification and Generation Method Based on BPMN

Jun Jin ¹ , Le Yan ¹, Yidan Zou ¹, Jie Li ^{1,2} and Zhen Yu ^{1,*}¹ School of Information, Beijing Wuzi University, Beijing 101149, China; jinjun@bwu.edu.cn (J.J.)² School of Computer Science and Engineering, Beihang University, Beijing 100191, China

* Correspondence: yz_bwu@163.com

Abstract: The post-deployment challenges in developing and upgrading blockchain smart contracts necessitate a high level of accuracy in their development and business logic. However, current methodologies for verifying the business logic of smart contracts frequently fail to address their alignment with end-user business requirements. This paper introduces a two-step language transformation process to bridge this gap. Initially, we establish a transformation rule from the Business Process Model and Notation (BPMN) to Prolog, enabling the translation of business processes into a Prolog representation. This step not only validates the business process logic but also ensures it meets user specifications. Subsequently, we introduce a transformation rule from the BPMN to Go, which facilitates the transformation of the BPMN model, once validated, into a Go language smart contract. To enhance usability, we have engineered a dedicated tool that streamlines this transformation process. We present a case study involving a banking loan process to exemplify the utility of our tool in creating BPMN diagrams, conducting requirement and syntax validations, and effecting the transformation to Go smart contracts. The case study and empirical results suggest that our methodology and the accompanying tool mitigate the complexities inherent in smart contract development. They also ensure the fidelity of business logic to user demands, thereby promoting the broader adoption of blockchain smart contract technology.

Keywords: BPMN modeling; business logic validation; user requirements verification; smart contract generation; language transformation consistency



Citation: Jin, J.; Yan, L.; Zou, Y.; Li, J.; Yu, Z. Research on Smart Contract Verification and Generation Method Based on BPMN. *Mathematics* **2024**, *12*, 2158. <https://doi.org/10.3390/math12142158>

Academic Editor: Florin Leon

Received: 12 April 2024

Revised: 29 May 2024

Accepted: 27 June 2024

Published: 10 July 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

MSC: 03B70

1. Introduction

A smart contract is a code script that operates within a blockchain network. Developers must utilize programming languages such as Solidity and Go, adhering to their respective syntax rules. As business requirements become increasingly complex, if developers do not have a good understanding of business logic, the implemented code logic is likely to be inconsistent with the business logic proposed by the user. Traditional business systems can be continuously corrected through version upgrades, but upgrading smart contracts after deployment is very difficult.

To reduce the difficulty of developing smart contracts, improve their correctness, and ensure consistency with business requirements, extensive research has been conducted both domestically and internationally on visual business logic modeling, model validation, and the generation of accurate models for smart contracts. References [1–4] propose graphical development methods and tools for Solidity language smart contracts. References [4–8] propose the verification of the structure and behavior of business models and process diagrams, but only focus on the basic structure of their models or process diagrams themselves, without considering and verifying whether user requirements are consistent with the model. Petri nets are mainly used for system behavior analysis and property verification, and are especially suitable for large-scale concurrent systems [9]. Prolog is a

declarative programming language that uses pattern matching and logical reasoning to validate rules [10]. Therefore, Prolog is more suitable for determining whether the structure and functions provided by smart contracts meet user requirements.

As a widely used business process modeling tool, a standard BPMN provides businesses with the capability of understanding their internal business procedures in a graphical notation and gives organizations the ability to communicate these procedures in a standard manner. Furthermore, the graphical notation facilitates the understanding of the performance collaborations and business transactions between the organizations [11]. Therefore, this paper chooses a BPMN and first uses the BPMN to model the business process of smart contracts, proposes transformation rules from the BPMN (xml format [12]) to Prolog, and uses Prolog to verify whether the BPMN is legal and meets user needs. Then, a set of transformation rules from the BPMN to Go programming language is proposed to transform the validated BPMN into a Go language smart contract. Finally, a case study is presented to demonstrate how to use developed tools to complete the above process, and experiments are conducted to demonstrate that the time required for the above transformation process is reasonable. This paper makes the following contributions:

1. To verify the correctness of the BPMN and its alignment with business requirements, we propose a set of transformation rules from the BPMN to Prolog. The Swish [13] supports Prolog-based validation, which enables the detection of errors in both syntax and the business description within the BPMN models.
2. To generate a Go smart contract, we propose a set of transformation rules from the BPMN to Go programming language.
3. We propose a new development process model that encompasses the design of visual requirements, their verification, and the subsequent generation of smart contract code. Demonstrated through a typical bank loan assessment case and experiments, this development paradigm alleviates the communication challenges between requirement specifiers and smart contract developers, providing a valuable enhancement to the existing paradigms in software engineering.

The rest of this paper is organized as follows. Section 2 discusses related work on model verification and visualization technologies for smart contracts. In Section 3, we propose a set of transformation rules from the BPMN to Prolog and from the BPMN to Go. Accordingly, the soundness and completeness of these transformation rules are demonstrated. Section 4 presents a case study on how to use our developed tool to generate a bank loan assessment smart contract, along with experiments that show the transformation time to be relatively short, thus meeting the requirements for practical applications. Section 5 provides a summary and outlines future work.

2. Related Works

To ensure the normal operation of business processes, it is necessary to examine the structure and behavior of business models and process diagrams, including organizational, resource, functional, and data modeling, to ensure that type attributes, associations, and instance logic are correct for static structures. For dynamic behavior, it is necessary to check the state transition and semantic tracking logic [14].

However, with the increasing complexity of business models, it is difficult to ensure the correctness of the models. So, we need to validate the model. Yamasathien S et al. [15] used Petri nets to model time business processes and discussed their constraint attributes, but did not provide a validation process or address how to handle constraint violations. Du Y H and Xiong P C et al. [16] extended Petri nets based on time and established relevant business processes. Then, they used model checking to verify the properties of the UPPAAL tool. However, the time complexity of the generated graph constructed by this method is not satisfactory, and its overall constraint consistency does not meet the requirements.

Prolog can describe complex information and has excellent descriptive and logical reasoning abilities, which enable it to effectively describe and validate business processes. Therefore, this paper conducts research based on the business process model BPMN and

selects Prolog as the business process description language for the verification of business process consistency.

In recent years, researchers have proposed new visualization technologies to help professionals better develop smart contracts. At present, smart contract generation is mainly aimed at Solidity contracts, such as the open source tool Caterpillar jointly developed by López Pinado O et al. [3], which can build, monitor, and optimize business processes through graphical methods, including graph transformation and generating Solidity smart contracts through BPMN-to-Solidity. Tran A B et al. [4] developed a smart contract visualization generation tool called Lorikeet based on Caterpillar, which can generate smart contracts based on the requirements of the business process model BPMN and registration data system through the BPMN to Policy transformation algorithm.

However, using these tools to graphically generate smart contracts lacks the necessary business logic verification process, and deploying them to blockchain without reliable verification may lead to security, demand, and other issues. Therefore, it is necessary to first verify the business logic of the visual smart contract. Then, in response to the many differences in the syntax between the business flow language BPMN and Go, a transformation rule for BPMN2Go was defined to achieve the transformation from XML-based BPMN models to Go smart contracts.

3. Smart Contract Modeling, Validation, and Generation

3.1. Method Overview

In order to validate the BPMN model and generate Go language smart contracts, this paper proposes the transformation from the BPMN flowchart to Prolog statements, two types of validation, and the transformation rules from the BPMN to Go language. As shown in Figure 1, the overall process is divided into four steps. ① Use BPMN notations on a graphical interface to model business logic as a BPMN model; ② Transform the BPMN model to Prolog statements and validate the basic rules of the BPMN model; ③ Verify whether the BPMN is legal and consistent with the user’s business requirements based on basic rules and customized business requirement rules; ④ According to the proposed transformation rules from the BPMN to Go language, transform the BPMN model into a smart contract.

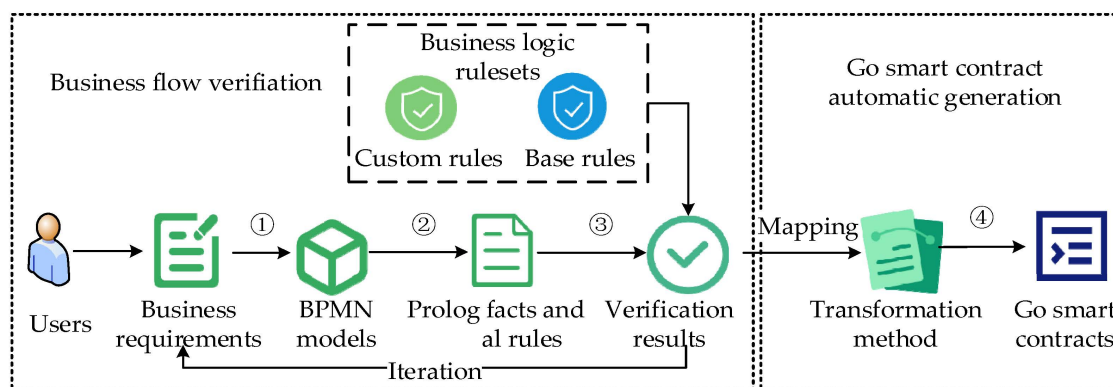


Figure 1. Framework for smart contract design and verification method.

3.2. Rules Based on Sequent Calculus

Sequent is a deductive formal statement [17] that can be represented by H as a premise, consisting of a finite number of predicates, and G as a target, consisting of a single predicate. The semantics of Formula (1) is that the target G holds if the premise H holds.

$$H \vdash G \tag{1}$$

Inference rules are used to generate proofs of sequents, consisting of two parts: the antecedent and consequent. The antecedent represents a finite set of sequents, while the

consequent represents a single sequent. An inference rule $R1$, which includes the antecedent A and consequent C , can be written in the following form. Its semantics is the following: as long as there is a proof of all the successive equations in antecedent A , inference rule $R1$ generates a sequent proof of consequent C . Using $\langle \rangle$ to represent the derivation process of program semantics, the general form is shown in Formula (2).

$$\langle \frac{A}{C} \rangle R1 \tag{2}$$

When the antecedent is an empty set denoted as \bullet , it can be written in the form of Formula (3) with the semantics that the inference rule $R2$ produces as a proof of the sequent C .

$$\langle \frac{\bullet}{C} \rangle R2 \tag{3}$$

The proof of a sequent can be seen as a finite tree, with each node in the form of (s, r) , consisting of a sequent s and an inference rule r . And the consequent of r is s . The sequents of all child nodes of this node (s, r) constitute the antecedent of r . For example, the following inference rules are given in Formulas (4)–(10).

$$\langle \frac{\bullet}{S2} \rangle R1 \tag{4}$$

$$\langle \frac{S7}{S4} \rangle R2 \tag{5}$$

$$\langle \frac{S2 \ S3 \ S4}{S1} \rangle R3 \tag{6}$$

$$\langle \frac{\bullet}{S5} \rangle R4 \tag{7}$$

$$\langle \frac{S5 \ S6}{S3} \rangle R5 \tag{8}$$

$$\langle \frac{\bullet}{S6} \rangle R6 \tag{9}$$

$$\langle \frac{\bullet}{S7} \rangle R7 \tag{10}$$

The proof of $S1$ is shown in Figure 2. The root node is $(S1, R3)$ and $S1$ is the expected sequent proof, which is also the consequent of $R3$. For a node such as $(S3, R5)$, its two child nodes include $S5$ and $S6$, which are the antecedents of $R5$. The semantics of this proof tree is the following: in order to prove $S1$, according to $R3$, it is necessary to prove $S2$, $S3$, and $S4$. Because $S2$ and $R1$ have no antecedents, it can be directly proven. By following this example, it can be proven that other sequences are used in the inference rule, which can be used to construct the transformation from child elements to the entire program through the setting of multiple rules.

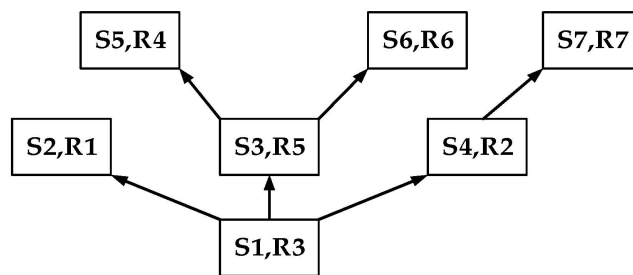


Figure 2. Sequent proof tree.

3.3. Definition of Transformation Operators

To describe the transformation, the three programming languages are represented as a five-tuple automaton $M = (Q, \Sigma, \delta, q_0, F)$, where Q represents the finite state set described by the language, δ represents the transition function, and q_0 and F represent the initial and ending state sets, respectively. Formulas (11)–(13) show the BPMN, Prolog, and Go language automata, respectively.

$$M_b = (Q_b, \Sigma_b, \delta_b, q_{0b}, F_b) \quad (11)$$

$$M_p = (Q_p, \Sigma_p, \delta_p, q_{0p}, F_p) \quad (12)$$

$$M_g = (Q_g, \Sigma_g, \delta_g, q_{0g}, F_g) \quad (13)$$

The transformation of language is essentially the transformation between state machines. The state machine of the BPMN is transformed into a Prolog automaton through transformation rules, and the process from the BPMN to Go is similar. Next, define the operators to describe the above transformation rules. From M_b to M_p and from M_b to M_g , there exists semantic consistency in the state sets of the three automata before and after the transformation.

1. Definition 1 (transformation operator φ)

The transformation rules for transforming from the BPMN to Prolog language, i.e., $M_p = \varphi(M_b)$.

2. Definition 2 (transformation operator ψ)

The transformation rules for transforming from the BPMN to Go language, i.e., $M_g = \psi(M_b)$.

3. Characteristic 1 (semantic consistency in language transformation)

If the state set and state transition function of the source language state machine and the target language state machine can be consistent through the transformation operator, it can be considered that the transformation process of the two languages has semantic consistency.

Taking the task element in the BPMN as an example, designing the transition rule as two types of corresponding language state transitions, using a sequent expression, can yield Formulas (14) and (15).

$$\left\langle \frac{\text{BPMN2Prolog}(bpmn2 \vdash \text{task } id \vdash \text{TaskID } name \vdash \text{TaskName})}{\text{task}(\text{TaskName}, \text{TaskId})} \right\rangle \text{task} \quad (14)$$

$$\left\langle \frac{\text{BPMN2Go}(bpmn2 \vdash \text{task } id \vdash \text{TaskID } name \vdash \text{TaskName})}{\text{func}(t * (\text{Structure } name)) \text{TaskName}} \right\rangle \text{task} \quad (15)$$






3.4. Smart Contract Modeling Based on BPMN

By using the BPMN to model the business logic of smart contracts, nodes form the foundation of the business process model. These nodes encompass tasks, events, network connections, and flow connections. Therefore, using the BPMN can easily build various different business process models. According to the BPMN 2.0 requirements, a process model consists of a series of notations, including the start event, end event, task, exclusive gateway, and business flow, typically saved in XML format. The BPMN has a wider application in the field of smart contracts, especially in multiple industries such as finance, credit, and asset trading that have strict security requirements.

Firstly, the user-proposed business logic is modeled using the BPMN (step ① in Figure 1). Generally, the program execution process can be categorized into three types: sequential execution, selective execution, and parallel execution. Due to the special processing required to ensure a consistent state across all nodes for the parallel execution of

blockchain smart contracts [18], this paper temporarily does not consider the transformation of parallel gateway nodes in the BPMN. Instead, it focuses on modeling start nodes, end nodes, task nodes, exclusive gateways, and sequence flows, as shown in Table 1.

Table 1. BPMN model notation.

| Name | Description | Symbol | Name | Description | Symbol |
|---------------|---|---|-------------------|--|---|
| start | The starting point of the process, defines how the process starts |  | exclusive gateway | Modeling decisions in a process |  |
| task | An atomic activity that represents an action that needs to be performed |  | end | The end of a branch in a process or subprocess |  |
| Sequence flow | A connector between two notations in a process |  | | | |

3.5. Transformation and Validation of BPMN to Prolog

Prolog is a logical language mainly used for the rapid validation of business processes. By formulating rules, it can achieve the validation of the BPMN models. Its automation level is high, and it can not only use backtracking and a non-deterministic search to explore possible solutions but also accurately describe business logic and inference validation. The formulation of validation rules needs to consider two aspects: BPMN model attributes and business scenarios; the former is the basic rule of the BPMN's own structure, used to check whether the BPMN model is legal. For example, a BPMN model only has one start event and one end event; the latter is a custom rule written based on specific business scenarios and logic. For example, in a bank lending scenario, users need to verify whether the existing model does not grant loans to people with low credit, and can customize rules according to their needs. Prolog can verify both types of rules mentioned above.

Next, transform the BPMN flowchart into Prolog language description, and simultaneously generate structural basic rules that can be used for validation (step ② in Figure 1).

Therefore, this paper proposes six transformation rules from the BPMN to Prolog, as outlined in Table 2. Specifically, rules 1 and 2 pertain to the transformation of the start and end nodes within the BPMN model. StartEventId represents the identifier of the start node, while StartEventName denotes the name of the start node. Rule 3 is the transformation of the task nodes in the BPMN model, where TaskId is the task node id and TaskName is the end node name. Rule 4 is the transformation of business flows in the BPMN model, where FlowId is the business flow ID, SourceId is the starting node ID of the business flow connection, and TargetId is the pointing node ID of the business flow connection. Rule 5 is the transformation of mesh nodes in the BPMN model, where GatewayName is the gateway node name and GatewayId is the gateway node ID. The BPMN model has been transformed into the basic rules for Prolog and the BPMN. During verification, in conjunction with the business requirement rules written by the user, use Prolog to determine whether the BPMN structure is legal and meets the business requirements proposed by the user (step③in Figure 1).

The algorithm for transforming the BPMN to Prolog statements is shown in Algorithm 1. The algorithm traverses and evaluates each element within the input BPMN file *bfile*. It employs two structures: *elist* and *flist*. The *elist* retains all critical notation elements, and any element not in *elist* is created and added upon encounter, as illustrated in lines 3, 6, 13, etc. The *flist* is utilized to store 'flow pairs' consisting of a node element and a flow element that have not yet established connections with two nodes, as demonstrated in lines 4, 9, 15, etc. For elements of type startEvent, task, endEvent, and exclusiveGateway, once added to *elist*, corresponding Prolog statements are generated and written to the Prolog file *plfile*, as illustrated in lines 3, 7, 13, etc. If a flow element *flowX* in *flist* matches the incoming or outgoing value of an element, the flow pair containing *flowX* is extracted, concatenated with the element to form a connectionLink structure, and then outputted to the *plfile*, as shown in lines 5, 12, 18,

etc. For elements of type sequence flow, if a flow pair composed of the element exists in *flist* and the sourceRef or targetRef of the element is present in *elist*, that flow pair is retrieved, and the flow element is concatenated with sourceRef and targetRef to generate a connectionLink structure, which is subsequently outputted to the *plfile*, as shown in line 22. If there is complex process logic (such as loop parallelism) in the BPMN process, or if the data objects and data mappings are incompatible with the data representation of the Prolog, inconsistent versions and other issues can lead to transformation failure.

Table 2. Transformation rules from BPMN to Prolog statements.

| BPMN Statement | Prolog Statement |
|---|--|
| <bpmn2:startEvent id="StartEventId" name="StartEventName"> | startNode(StartEventName). |
| <bpmn2:endEvent id="EndEventId" name="EndEventName"> | endNode(EndEventName). |
| <bpmn2:task id="TaskId" name="TaskName"> | task(TaskName). taskName(TaskName,TaskId). |
| <bpmn2:sequenceFlow id="FlowId" sourceRef="SourceId" targetRef="TargetId" /> | connectionLink(FlowId,SourceId,TargetId). |
| <bpmn2:exclusiveGateway id="GatewayId" name="GatewayName"> | gateway(GatewayName). gatewayName(GatewayName,GatewayId). |

Algorithm 1: BPMN to Prolog Statements Transformation

Input: BPMN file *bfile* (.xml)

Output: Prolog statement description file *plfile* (.pl)

```

1  Traverse all notations in bfile until the end;
2  switch (type of the notation)
3    case startEvent: startX → elist; startNode(StartEventName). → plfile ;
4      if (flowOutY ∈ elist && nodeZ ∈ elist) then <flowOutY,nodeZ > → flist;
5      else <flowOutY,nodeZ > ← flist; connectionLink(flowOutY, startX, nodeZ). → plfile;
6    case exclusiveGateway: exGateX → elist;
7      gateway(GatewayName). → plfile; gatewayName(GatewayName,GatewayId). → plfile;
8      if (flowInY ∈ elist || flowOutZ ∈ elist) then
9        <flowInY, exGateX> → flist; <flowOutZ, exGateX > → flist;
10     else <flowX, nodeN> ← flist;
11     if (flowX == flowInY || flowX == flowOutZ) then
12       connectionLink(flowInY, nodeN, exGateX). || connectionLink(flowOutZ, exGateX, nodeN). → plfile;
13    case task: taskN → elist ; task(TaskName). → plfile; taskName(TaskName,TaskId). → plfile;
14     if (flowInY ∈ elist && flowOutZ ∈ elist) then
15       flowInY && flowOutZ → elist; <flowInY, taskN> && <flowOutZ, taskN> → flist;
16     else <flowX, taskN> ← flist;
17     if (flowX == flowInY || flowX == flowOutZ) then
18       connectionLink(flowInY, taskN, nodeN). or connectionLink(flowOutZ, nodeN, taskN). → plfile;
19    case sequence flow: if (flowX ∈ elist) then flowX → elist;
20     if (sourceRefX ∈ elist && targetRefX ∈ elist) then
21       <flowX, sourceRef> ← flist; <flowX, targetRef> ← flist;
22       connectionLink(flowX, sourceRefX, targetRefX). → plfile;
23     else if (sourceRefX ∈ elist || targetRefX ∈ elist) then
24       sourceRefX || targetRefX → elist; <flowX, sourceRefX> || <flowX, targetRefX> → flist;
25    case endEvent: endN → elist; endNode(EndEventName). → plfile;
26     if (flowInY ∈ elist) then
27       flowInY → elist ; <flowInY, endN> → flist;
28     else <flowInY, nodeX> ← flist;
29     connectionLink(flowInY, nodeX, endN). → plfile;
30  end switch
31  Output: plfile

```

The basic rules of the BPMN model and the user-defined rules are described using Prolog statements. If the input BPMN model does not meet the above rules, it cannot pass validation. The verification process is shown in Algorithm 2, which includes eight rules. The first rule is used to verify that there is no connection between all start nodes. The second clause is similar to the first clause, and is used to verify that there is no connection between all endpoints. Line 3: Verify that there is no connection between any start and end nodes. Lines 4 and 5, respectively, verify that all start and end nodes are not isolated nodes and must have connections with other non-start or end nodes. Line 6: Verify that all gateway nodes are not isolated nodes and must be the starting or ending nodes of a connection. Line 7: The other end of the connection where the gateway node is located must be a task node and cannot be another gateway node (the BPMN2.0 does not allow two exclusive gateways to be connected). The eighth validation checks whether there are node pairs in the given task node pair (Tasknodes) that are not connected.

Algorithm 2: BPMN Verification

Input: BPMN file (.xml)

Output: Correctness

```

1  IsAnyEndnodePairNotConnected()
2  IsAnyStartnodePairNotConnected()
3  IsAnyStartEndNodePairNotConnected(Startnode,Endnode)
4  IsStartnodeConnctected(Startnodes)
5  IsEndnodeConnctected(Endnodes)
6  IsAllGatewaynodesConnected(Gatewaynodes)
7  IsGatewayandTaskNodesConnected(Tasknode1,Tasknode2,Gatewaynode)
8  IsAnyTasknodePairNotConnected(Tasknodes)

```

3.6. BPMN Transformation and Smart Contract Generation

After the BPMN model has been verified, the transformation to a Go language smart contract is realized in accordance with the semantic consistency between the BPMN model and logical and imperative languages such as smart contracts (step ④ in Figure 1).

To achieve this, this paper establishes transformation rules for Go smart contracts based on the five types of notations present in the BPMN model, as illustrated in Table 3. This enables the generation of Go smart contracts by transforming each notation individually.

Rule 1: Transformation of the BPMN model root notation definitions, where Contract-Name is the name of the BPMN model and becomes the name of the Go smart contract after transformation.

Rule 2: The transformation of the BPMN business process task nodes involves transforming their name, id, and constraints into Go smart contract functions. Additionally, it is important to note that the XML file of a BPMN business process must consist of at least one task node.

Rule 3: The third rule involves the transformation of the start node in the BPMN file, known as startEvent, into the main function of the Go smart contract. This includes translating the name, id, and constraints utilized to construct the BPMN model. Typically, the XML file of a BPMN contains just one line of code that represents the startEvent notation as an entry function, which is analogous to the main function in the Go smart contract code.

Rule 4: The BPMN business process sequence flow notation, namely sequenceFlow, along with its attributes id, sourceRef, targetRef, and constraints, is translated into a call relationship between functions within the Go smart contract. The sourceRef attribute denotes the starting point of the sequence flow, while the targetRef attribute signifies the endpoint of the sequence stream. Consequently, the TargetId function is executed subsequent to the function whose task ID is SourceId.

Rule 5: The BPMN employs a transformation for the selection gateway notation. The selection gateway file in the BPMN can comprise several notations, such as id, name, and the constraints used to construct the BPMN model. Each BPMN gateway possesses two

sequences. If condition 1 is met, sequence flow 1 is executed; otherwise, sequence flow 2 is executed. This rule maps the judgment criteria for selecting a gateway in the BPMN model to a selection statement for the Go smart contract code.

Table 3. Notation mapping rules from BPMN to Go smart contracts.

| Index | BPMN Notation | Notations of Go Smart Contract |
|-------|---|--|
| 1 | <bpmn2:definitions name="ContractName" other notations..... </bpmn2:definitions> | ContractName.go |
| 2 | <bpmn2:task id="TaskID" name="TaskName"> other notations..... </bpmn2:task> | func TaskName(// Parameters for function) (return value, error){ task specific logic... } |
| 3 | <bpmn2:startEvent id=" StartEventId " name="StarteventName"> </bpmn2:startEvent> | func main(){ main function code... } |
| 4 | <bpmn2:sequenceFlow id="StartEventId" sourceRef="SourceId" targetRef="TargetId"> </bpmn2:sequenceFlow> | func SourceIdName(){ task function code corresponding to SourceId. . .} func TargetIdName(){ task function code corresponding to TargetId. . .} |
| 5 | <exclusiveGateway id="GatewayName"> <documentation> Determine whether condition 1 is met </documentation> </exclusiveGateway> <sequenceFlow id="flow1 " sourceRef="gate" targetRef="Target Node"> <![CDATA[Condition 1]]> </sequenceFlow> | if(Condition 1){ business function code... } else{ business function code... } |

Task node reuse rule: If multiple identical tasks exist in a BPMN model, their transformations do not need to be repeated. Instead, the transformed result can be directly reused. As illustrated in Figure 3, the BPMN model comprises four tasks, with three being TaskA and one being TaskB. In such a case, the transformation of TaskA is not subject to redundancy; post-transformation, the resultant Go language function can be successively invoked. Consequently, this rule can amplify the efficiency of smart contract generation.

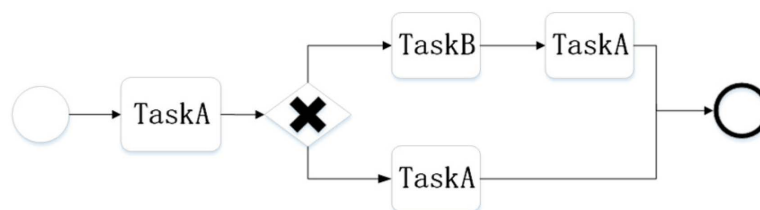


Figure 3. BPMN example of task transformation reuse.

3.7. Soundness and Completeness

3.7.1. Soundness

Although the BPMN is not an executable language, it still has precise definitions for notations. In the BPMN statement of Table 2, the definition of the task is expressed as Formula (16), and its core is $action_i$, which is used to describe the state transition action that needs to be executed. After receiving $input_i$ and executing $action_i$ with its name $taskName$,

the state changes from $state_{i-1}$ to the new $state_i$, provided that the prerequisite condition satisfies $preCondition_i$.

$$\text{task} ::= \frac{\text{input}_i(\text{taskName}, \text{state}_{i-1})\{\text{action}_i\} \wedge \text{preCondition}_i}{\text{action}_i \leadsto \text{state}_i} \quad (16)$$

For the Prolog statements after task transformation, it can be seen from the transformation of the two types of task and sequenceFlow in Table 2 that the corresponding Prolog contains both the preceding and succeeding tasks with TaskName and TaskID, and the relationship between the preceding and succeeding is determined by the two sequenceFlows in the inbound and outbound directions. Its actual semantics can be expressed as Formula (17), which represents the end state of the previous task, $TargetId_{i-1}$ or $SourceId_i$, which is the initial state of this task. It should be consistent with the initial state of the task corresponding to the transformed prolog statements, that is, $TargetId_{i-1} = state_{i-1}$ ($state_{i-1}$ that appears in Formula (16)) and their $action_i$ must perform the same operation. According to the principle of Turing machines, the same input and execution process will inevitably get the same result, that is, $TargetId_i$ and $state_i$ that appears in Formula (16) are equivalent.

$$\text{taskName} ::= \frac{TargetId_{i-1}}{\text{action}_i \leadsto TargetId_i} \quad (17)$$

Similarly, the semantics of the corresponding Go language in Table 3 can be expressed as Formula (18).

$$\text{function} ::= \frac{\text{functionName}(\text{state}_{i-1}, \text{action}_i)}{\text{action}_i \leadsto \text{newState}_i} \quad (18)$$

In summary, since Go and the BPMN both have $state_{i-1}$ as the initial state and $action_i$ as the execution process, the results obtained should also be consistent, that is, $TargetId_i = state_i$ and $\text{newState}_i = state_i$. It can be assumed that if the input for each step is the same and the user implements the same state transition process, the results obtained for the three languages must be the same. For other notations such as gateways, they also have similar properties. It can be seen that this line-by-line language transformation method for different notations can achieve consistent logic and can be considered to have soundness.

3.7.2. Completeness

Considering the practical characteristics of smart contracts, the transformation in this article does not cover all notations of the BPMN language. For example, due to the issue that smart contracts cannot be executed in parallel, the definition and transformation of parallel gateways and Merge gateways were not considered. The core notations, such as task, flow, exclusiveGateway, start event, etc., can be used to cover the complete functional descriptions of smart contract functions, branch statements, beginning and ending statements of the contract, etc. Guided by practical requirements, removing unnecessary notation definitions helps reduce the code for transformation and also standardizes user input.

4. Experimental Analysis

4.1. Application Cases

Smart contract verification and generation are applied in numerous fields, particularly in finance, credit, asset trading, and other sectors with stringent security requirements. This paper demonstrates how users can employ the implemented tools, ranging from BPMN modeling to generating Go smart contracts, using a bank loan assessment application as an example.

First, the user proposes the business requirements for a loan application, assessment and issuance, using the BPMN modeling function in the tool, as shown in Figure 4. The process involves a lender applying for a loan at the bank, where the bank conducts a credit assessment based on the lender's credit status and information provided. The bank gives the loan plan according to the credit evaluation: no loan if the evaluation fails (credit less

than 60), a petty loan (credit greater than or equal to 60 and less than 80), or a large loan (credit greater than or equal to 80). If the credit assessment is passed, the lender signs a loan contract with the bank, and then handles mortgage registration in accordance with relevant national laws and regulations, and finally the bank issues loans according to the assessment limit and relevant regulations.

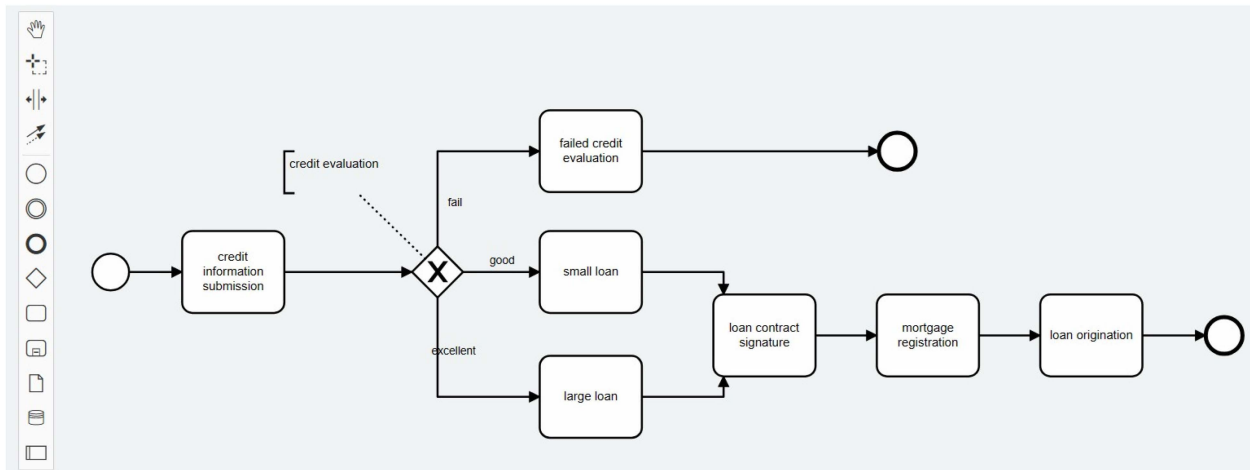


Figure 4. Model the bank loan assessment process as a BPMN diagram using the toolbar on the left side of the implemented modeling tool.

Next, the system validates the model, and users can enter custom rules. In a Prolog statement, the symbol “:-” is the clause indicator, used to distinguish the head and body. The symbol “\+” is the negation operator, representing “not” or “does not have”. For example, to determine whether the BPMN process defines a person with a credit below 60 who cannot pass loan approval, the custom rule section can be entered:

```
cannot_loan(Person):-
    credit_score(Person, Score), Score < 60;
```

```
To query whether a loan can be made, the custom rule is:
loan_eligibility(Person):- \+cannot_loan(Person).
```

After the BPMN model is validated, a Go language smart contract is generated through the transformation rules in Table 3, with some of the code shown in Figure 5.

| BPMN | Code generation |
|---|---|
| <p>here is the BPMN file content that you selected</p> <pre> 1 <?xml version="1.0" encoding="UTF-8"?> 2 <bpmn2:definitions xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:bpmn2="http://www.omg.org/ 3 <bpmn2:process id="Process_1" isExecutable="false"> 4 <bpmn2:startEvent id="StartEvent_1"> 5 <bpmn2:outgoing>Flow_lq22an2</bpmn2:outgoing> 6 </bpmn2:startEvent> 7 <bpmn2:task id="Activity_1n609x2" name="provide credit information"> 8 <bpmn2:incoming>Flow_lq22an2</bpmn2:incoming> 9 <bpmn2:outgoing>Flow_l2wva48</bpmn2:outgoing> 10 </bpmn2:task> 11 <bpmn2:sequenceFlow id="Flow_lq22an2" sourceRef="StartEvent_1" targetRef="Activity_1n609x2" /> 12 <bpmn2:exclusiveGateway id="Gateway_0rvuxd6"> 13 <bpmn2:incoming>Flow_l2wva48</bpmn2:incoming> 14 <bpmn2:outgoing>Flow_0o58h54</bpmn2:outgoing> 15 <bpmn2:outgoing>Flow_0f1r4q8</bpmn2:outgoing> 16 <bpmn2:outgoing>Flow_0996x14</bpmn2:outgoing> 17 </bpmn2:exclusiveGateway> 18 <bpmn2:task id="Activity_0oyvgs1" name="fail credit assessment"> 19 <bpmn2:incoming>Flow_0o58h54</bpmn2:incoming> 20 <bpmn2:outgoing>Flow_lmx29s</bpmn2:outgoing> 21 </bpmn2:task> 22 <bpmn2:task id="Activity_04630n7" name="small loan"> 23 <bpmn2:incoming>Flow_0f1r4q8</bpmn2:incoming> 24 <bpmn2:outgoing>Flow_lq4dnpk</bpmn2:outgoing> 25 </bpmn2:task> 26 <bpmn2:task id="Activity_19w0i3c" name="jumbo loan"> 27 <bpmn2:incoming>Flow_0996x14</bpmn2:incoming> 28 <bpmn2:outgoing>Flow_lq4dnpk</bpmn2:outgoing> 29 </bpmn2:task> 30 <bpmn2:sequenceFlow id="Flow_0o58h54" name="fail the assessment" sourceRef="Gateway_0rvuxd6" targ 31 <bpmn2:sequenceFlow id="Flow_0f1r4q8" name="assessed well" sourceRef="Gateway_0rvuxd6" targetRef= 32 <bpmn2:sequenceFlow id="Flow_0996x14" name="evaluate excellence" sourceRef="Gateway_0rvuxd6" targ 33 <bpmn2:endEvent id="Event_0d6z4ie"> 34 <bpmn2:incoming>Flow_lmx29s</bpmn2:incoming> 35 </bpmn2:endEvent> 36 <bpmn2:sequenceFlow id="Flow_lmx29s" sourceRef="Activity_0oyvgs1" targetRef="Event_0d6z4ie" /> 37 <bpmn2:task id="Activity_14h2fzr" name="sign a loan contract"> 38 <bpmn2:incoming>Flow_lq4dnpk</bpmn2:incoming> 39 <bpmn2:outgoing>Flow_lq1qtr8</bpmn2:outgoing> 40 </pre> | <p>here is the generated code content</p> <pre> 1 package main 2 import (3 "encoding/xml" 4 "fmt" 5 "io/ioutil" 6 "github.com/hyperledger/fabric-chaincode-go/shim" 7 pb "github.com/hyperledger/fabric-protos-go/peer" 8) 9 10 11 type Process struct { 12 StartEvent StartEvent `xml:"startEvent"` 13 Task Task `xml:"task"` 14 SequenceFlows []SequenceFlow `xml:"sequenceFlow"` 15 ExclusiveGateway ExclusiveGateway `xml:"exclusiveGateway"` 16 EndEvent EndEvent `xml:"endEvent"` 17 TextAnnotation TextAnnotation `xml:"textAnnotation"` 18 Association Association `xml:"association"` 19 } 20 21 type StartEvent struct { 22 ID string `xml:"id"` 23 Outgoing string `xml:"outgoing"` 24 } 25 26 type Task struct { 27 ID string `xml:"id"` 28 Name string `xml:"name"` 29 Incoming string `xml:"incoming"` 30 Outgoing string `xml:"outgoing"` 31 } 32 33 type SequenceFlow struct { 34 ID string `xml:"id"` 35 SourceRef string `xml:"sourceRef"` 36 TargetRef string `xml:"targetRef"` 37 } 38 39 type ExclusiveGateway struct { 40 ID string `xml:"id"` </pre> |

Figure 5. Go language smart contract for the bank loan assessment business process.

4.2. Experimental Analysis

The performance of key steps in business processes, including model transformation, validation, and smart contract generation, can be influenced by the simplicity or complexity of these processes. This section presents three experiments designed to measure the duration of model transformation. The first experiment involves transforming a BPMN model into Prolog statements and rules, while the second one focuses on transforming a BPMN model into a Go smart contract with repeated task nodes reused. The last one tests the time reduction effect of reusing duplicate task nodes. The experimental environment is configured with an Intel i5-1135 G7 CPU and a Windows 11 operating system.

4.2.1. Analysis of Model Transition Time from BPMN to Prolog

The transformation of the BPMN model to Prolog mainly lies in the transformation of task nodes and gateway nodes. To evaluate the transformation time for business processes, an experiment was conducted where the number of task nodes ranged from 10 to 50. This representation aimed to depict a progression from simple to complex business processes. The total time taken for model transformation and the average time per transformation were both measured and calculated.

The experiment tests the model transformation time from the BPMN to Prolog. Figure 6a,b, respectively, show the total time required for model transformation when the BPMN model contains 10–50 task nodes and 5–25 gateway nodes. Figure 6a shows a roughly linear relationship between the transformation time and the number of task nodes included within the model, indicating that the change in the number of task nodes does not affect the transformation time for a single task node. Meanwhile, Figure 6b shows an approximate square increase in the transformation time of the model as the number of network nodes increases. The primary reason is that during the transformation of the gateway node, it is essential to verify the correctness of the generated flow path. If the flow path is incorrect, the process must revert to the gateway node's branch point and conduct a conditional assessment or a backtracking review of the business logic involved.

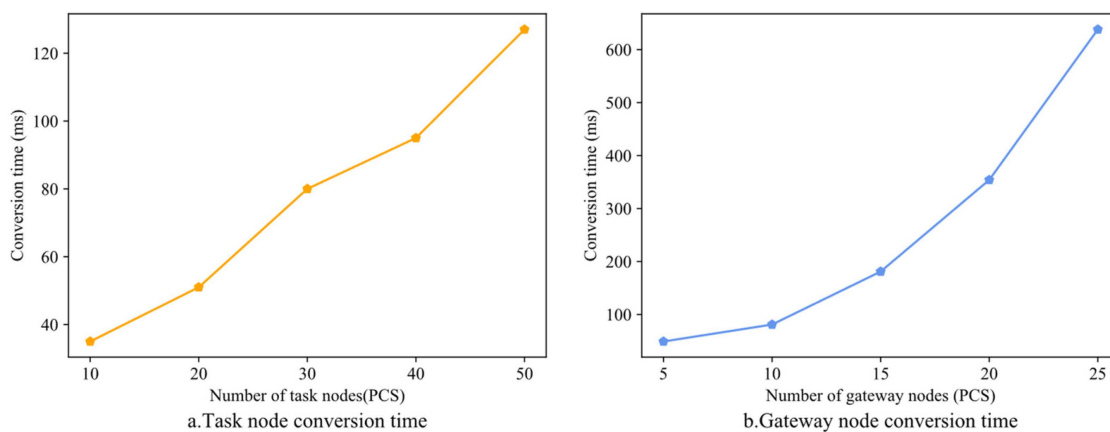


Figure 6. Transformation time of BPMN to Prolog statements.

In the transformation from the BPMN to Prolog, the time taken for transforming task nodes remains relatively constant. However, the transformation time for gateway nodes increases at a rate slightly faster than linear, in proportion to the increase in the number of gateway nodes. Nonetheless, for a straightforward business scenario with no more than 40 task nodes and 20 gateway nodes in the BPMN model, the transformation time from the BPMN model to the Prolog model is under 0.5 s. This meets the time requirement for business process verification. The proposed method for model transformation and verification from the BPMN to Prolog is thus deemed feasible.

4.2.2. Transition Time Analysis of BPMN to Go Language Smart Contract

This section evaluates the transformation time of BPMN models to the Go smart contracts, considering scenarios with 10–50 task nodes and gateways. The experimental results in Figure 7 show that the transformation time from the BPMN to Go language smart contract, similar to the case of transforming to Prolog, also increases with the number of task nodes and gateway nodes. If the number of task nodes exceed 50 or more, the reuse rules described in Section 3.6 can be employed to optimize the overall transformation time.

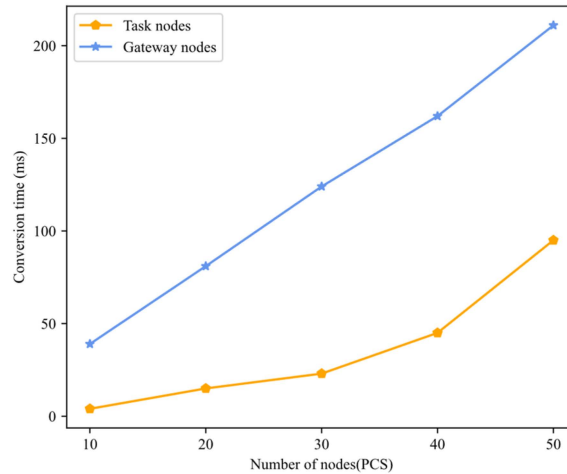


Figure 7. Transformation time of BPMN to Go smart contract.

4.2.3. Analysis of Transformation Time before and after Code Reuse

In the context of a large number of task nodes (e.g., more than 50) or multiple task nodes performing the same task, there are five scenarios where the proportion of identical task nodes in the total number of nodes within a BPMN process ranges from 52% to 60%: six identical nodes in 10 task nodes, 11 identical nodes in 20 task nodes, 16 identical nodes in 30 task nodes, 21 identical nodes in 40 task nodes, and 26 identical nodes in 50 task nodes are tested both before and after code reuse. In Figure 8, when the process contains 50 task nodes, the transformation time after code reuse is reduced by about 30% (from 95 ms to 63 ms) compared to before code reuse, indicating that the task node reuse rule proposed in Section 3.6 is effective.

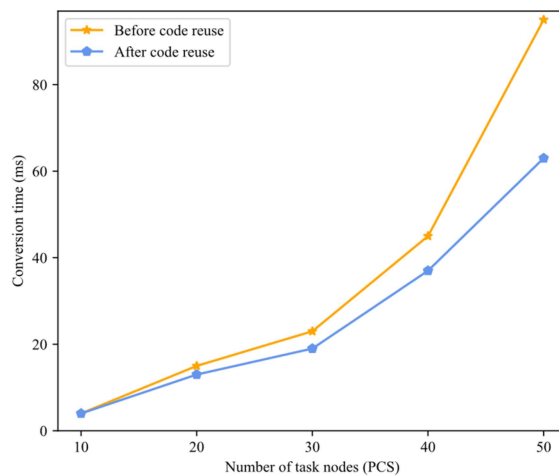


Figure 8. Comparison of transformation time before and after code reuse.

From the experimental results of the above three sections, it can be seen that under normal conditions where the application is not too complex, the number of task nodes and branch gateway nodes included in the model is not large. Therefore, the time required to

convert from the BPMN to Prolog will be within 0.5 s, and the time for conversion from the BPMN to Go language will be within 0.3 s. The entire process of verification and generation of Go language smart contracts will be completed within 1 s, which can meet the needs of most applications.

5. Conclusions

In this paper, we proposed the transformation rules from the BPMN to Prolog statements and from the BPMN to smart contract of Go programming language, as well as a business logic verification scheme based on the BPMN. Through formal description, case studies, and experiments, it has been proven that business logic verification can be achieved, and the transformation time from the BPMN to Prolog and Go smart contracts is within 1 s, which can meet the requirements of most practical applications. By using the above BPMN transformation rules and verification scheme, the user can effortlessly accomplish the design of requirements, the verification of these requirements, and the generation of a smart contract code. This novel development process paradigm mitigates communication impediments between requirement stakeholders and contract developers, effectively complementing the established practices in software engineering.

The current research is dedicated to the transformation of the BPMN models into the Go programming language. Subsequent endeavors are projected to broaden the scope to encompass advanced constructs in the BPMN and more smart contract languages, with the objective of optimizing the smart contract generation process.

Author Contributions: Methodology, J.J. and Z.Y.; Software, L.Y.; validation, J.L. and Z.Y.; writing—original draft, Y.Z.; writing—review and editing, J.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Beijing Natural Science Foundation (grant number M22040), the Youth Foundation of Beijing Wuzi University (grant number 2022XJQN24), the Science and Technique General Program of Beijing Municipal Commission of Education (grant number KM201910037003), the Research Base Project of Beijing Municipal Social Science Foundation (grant number 18JDGLB026).

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Graphical Smart Contract Generation Editor. Available online: <http://www.lianmenhu.com/blockchain-4658-1> (accessed on 16 December 2023).
2. Li, Y.S.; Li, Y.; Jin, X. Visual Blockchain Smart Contract Framework and Deployment Methods for Smart Contract Development. CN112148278A, 29 December 2020.
3. López-Pintado, O.; García-Bañuelos, L.; Dumas, M.; Weber, I. Caterpillar: A Blockchain-Based Business Process Management System. *Bus. Process Manag.* **2017**, *172*, 1–5.
4. Tran, A.B.; Lu, Q.; Weber, I. Lorikeet: A Model-Driven Engineering Tool for Blockchain-Based Business Process Execution and Asset Management. In Proceedings of the International Conference on Business Process Management, Sydney, NSW, Australia, 9–14 September 2018.
5. Gajski, D.D.; Abdi, S.; Gerstlauer, A.; Schirner, G. *Embedded System Design: Modeling, Synthesis and Verification*; Springer: New York, NY, USA, 2009; pp. 49–111.
6. Baier, C.; Katoen, J.P. *Principles of Model Checking*; MIT press: Cambridge, MA, USA, 2008; pp. 25–40.
7. Cosenz, F.; Rodrigues, V.P.; Rosati, F. Dynamic Business Modeling for Sustainability: Exploring a System Dynamics Perspective to Develop Sustainable Business Models. *Bus. Strateg. Environ.* **2020**, *29*, 651–664. [[CrossRef](#)]
8. Yu, W.S. Research on task-oriented business process modeling and verification method. Master's Thesis, Nanjing University of Aeronautics and Astronautics, Jiangsu, China, 2015.
9. Ghilardi, S.; Gianola, A.; Montali, M.; Rivkin, A. Petri Nets with Parameterised Data: Modelling and Verification. In Proceedings of the International Conference on Business Process Management, Seville, Spain, 13–18 September 2020.
10. Hermenegildo, M.V.; Morales, J.F.; Lopez-Garcia, P.; Carro, M. Types, Modes and so Much More—The Prolog Way. In *Prolog: The Next 50 Years*, 1st ed.; Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M.V., Kowalski, R., Rossi, F., Eds.; Springer Nature: Cham, Switzerland, 2023; Volume 13900, pp. 23–37.

11. Business Process Model and Notation. Available online: <http://www.bpmn.org/> (accessed on 14 May 2024).
12. Maqbool, B.; Azam, F.; Anwar, M.W.; Butt, W.H.; Zeb, J.; Zafar, I.; Nazir, A.K.; Umair, Z. A Comprehensive Investigation of BPMN Models Generation from Textual Requirements—Techniques, Tools and Trends. In Proceedings of the Information Science and Applications (ICISA), Hong Kong, China, 20–22 March 2018.
13. Swish. Available online: <https://www.swi-prolog.org/> (accessed on 14 May 2024).
14. Liu, Y.; Ma, Z.Y.; He, X.; Shao, W.Z. A conversion method from UML model to reliability analysis model. *J. Softw.* **2010**, *21*, 287–304. [[CrossRef](#)]
15. Yamasathien, S.; Vatanawood, W. An Approach to Construct Formal Model of Business Process Model from BPMN Workflow Patterns. In Proceedings of the 2014 Fourth International Conference on Digital Information and Communication Technology and its Applications (DICTAP), Bangkok, Thailand, 6–8 May 2014.
16. Du, Y.; Xiong, P.; Fan, Y.; Li, X. Dynamic Checking and Solution to Temporal Violations in Concurrent Workflow Processes. *IEEE Trans. Syst. Hum. Cybern. A* **2011**, *41*, 1166–1181.
17. Downen, P.; Ariola, Z.M. A Tutorial on Computational Classical Logic and the Sequent Calculus. *J. Funct. Program* **2018**, *28*, e3. [[CrossRef](#)]
18. Shi, J.F.; Wu, H.; Gao, H.R.; Zhang, W.B. Overview on Parallel Execution Models of Smart Contract Transactions in Blockchains. *J. Softw.* **2022**, *33*, 4084–4106.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.