Smart Contract Generation Supporting Multi-instance for Inter-Organizational Process Collaboration

Shangqing Feng[1], Chang Jia[1], Maolin Pan[2], and Yang Yu[2](✉)

[1] School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China
{fengshq5,jiach}@mail2.sysu.edu.cn
[2] School of Software Engineering, Sun Yat-sen University, Zhuhai, China
{panml,yuy}@mail.sysu.edu.cn

Abstract. Blockchain can help to maintain trusted Inter-Organizational Process Collaboration (IOPC) among parties or organizations. The key to such studies is to translate the IOPC models into smart contracts, the executable programs in the blockchain. There are some smart contract generation solutions for IOPC. However, most of them don't support multi-instance, which is an important feature of IOPC. The proposed method is designed to fill the research gap. One core of our method is that it sums up multi-instance elements of BPMN collaboration diagram and use Communicating Sequential Programs (CSP#) to formalize them. This formalization focuses on interactions, helps to get precise process execution semantics and makes model verification feasible, avoiding flawed models being translated. The other core is a translation technique based on syntax trees to generate smart contracts from CSP# models. The method is implemented and evaluated in terms of features and cost.

Keywords: Inter-Organizational Process Collaboration · Multiple instance · BPMN · Verification · Smart Contract.

1 Introduction

With the development of economic globalization, inter-organizational process collaboration (IOPC) has become more and more common [1]. Because the participants[3] of IOPC are from different parties or organizations, the traditional centralized business process collaboration cannot prevent participants from cheating, resulting in a lack of trust [2]. Blockchain technology emerging in recent years provides an idea to solve this problem [2,3]. As a decentralized distributed ledger, blockchain can establish trusted data sharing among the organizations and make sure the process execution records are tamper-proof [4,5]. The core and premise of blockchain-based IOPC is to translate IOPC models (e.g., written in BPMN) to smart contracts.

[3] In this paper, a participant is a representative of an organization. And sometimes, it refers to an organization.

Some solutions for smart contract generation are provided as a part of tools such as Caterpillar [5], Lorikeet [6] and ChorChain [7]. Some are proposed for specific goals such as cost optimization [2, 8] and dealing with time-dependent events [9]. However, most of them don't provide support for multi-instance, which is an important feature of IOPC. Multi-instance means that some tasks in a business process need to be executed by several participants in parallel, or that multiple participants execute the same process (but different instances) within the same period. It is frequent in IOPC. For example, when reviewing a paper, multiple participants play the role of *Reviewer* to do the same tasks independently in the same period. In this context, existing solutions no longer work. In order to expand the scope of application, it is necessary for blockchain-based IOPC to support multi-instance. In other words, it is important to propose a smart contract generation method that supports multi-instance IOPC.

To achieve the goal of transforming multi-instance IOPC models to smart contracts, there are some problems and challenges: (1) Different from the ordinary elements, multi-instance elements require new processing logic. For example, a multi-instance task can have multiple instances, each of which is similar to an ordinary task. But every instance is considered part of the task and must conform to the constraints imposed on the task. (2) There are many multi-instance elements and their meanings vary in detail. These elements should be distinguished and translated to different contract code. (3) To avoid flawed models being translated and deployed, an approach to verify the models in advance is needed. Multi-instance complicates the execution of IOPC. The more complex the situation, the more likely the model contains flaws (like deadlock in the control flow). This will make the generated contract flawed, too. Once deployed, it may cause huge loss.

There are some studies that can help us resolve the challenges. [10] has provided a Backus-Naur Form (BNF) syntax of BPMN collaboration structure. Based on the BNF syntax, [11] uses Communicating Sequential Programs (CSP#) [12], which supports interaction description and verification well, to formalize a part of core elements in BPMN collaborations. In this paper, we summarize the interaction elements in BPMN collaborations so that we can cover most of the multi-instance IOPC. Then, we extend the CSP# formalization in [11] to support all of the elements, so that we can verify multi-instance IOPC. To get the final contract, a technique based on syntax trees is developed to generate smart contracts from CSP# models.

To sum up, the major contributions in this paper are:

- We provide a summary of interaction elements in BPMN collaborations involving multi-instance, a CSP# formalization for the elements, and a definition of soundness of CSP# models in this context. These helps to verify multi-instance IOPC from interactions perspective.
- We develop a translation technique to generate smart contracts from CSP# models. This technique captures the logic order of the CSP# processes based on syntax trees and generate smart contract code for each atomic CSP# process.

- We propose a low code method that supports model verification and automatic smart contract generation for multi-instance IOPC. The method takes a BPMN collaboration model as input. It first transforms the collaboration model to a CSP# model, then verifies the CSP# with the soundness definition and an existing CSP# analysis tool. If the CSP# model passes the verification, it generates smart contract code based on the CSP# model.

The rest of this paper is organized as follows: Section 2 reviews the related work. Section 3 lists the summarized interaction elements and shows a running example to illustrate the context. The detail of the proposed contract generation method is described in Section 4. Section 5 holds evaluation, followed by conclusion and future work in Section 6.

2 Related Work

This section provides an overview of the relevant literature, focusing on studies that are particularly relevant to our work.

2.1 Research on Smart Contract Generation related to IOPC

Recent research related to blockchain-based IOPC and smart contract generation has focused on providing a business process management system, or reducing the cost of smart contracts. Most of the research work does not provide support for multi-instance IOPC.

Weber et al. [4] presented the first blockchain-based approach and implementation to address the lack of trust in IOPC, in which a translator was designed to derive a smart contract from a BPMN choreography model. It generates smart contract code according to the patterns of choreography elements. But due to the lack of model vedrification, if there are flaws in the choreography, it is possible that the contract is unsuitable to be deployed.

After that, García et al. [8] proposed a method to optimize the cost for executing business processes on top of blockchain. The method translates the BPMN process model into a Petri Net, and generates a smart contract in Solidity based on the reduced Petri Net. A similar approach was introduced by Nakamura et al. in [2]. The difference is that [2] used the BPMN process model with multiple lanes to model IOPC and used state charts for optimization work.

López-Pintado et al. [5] introduced a completely blockchain-based BPMS execution engine named Caterpillar, whose compiler is based on the translator of [4]. Tonga Naha et al. [9] extended Caterpillar to support time-dependent events and inclusive gateways

The methods proposed in [8], [2], [5] and [9] focus on dealing with BPMN process diagrams. They require the IOPC to be modelled as a BPMN process diagram, like a process within an organization. However, different from the process within an organization, an important focus of IOPC is the interactions (that is message exchanges) among the participants/organizations, which indicate how multiple participants/organizations cooperate to achieve the shared goal [10]. While the BPMN process diagram has no advantages in describing these. In other words, these methods don't support two important features of IOPC—multi-instance and interaction description.

Note that, in [5], all the participants share the same workflow engine—blockchain, and execute the whole process (including the inner tasks) in blockchain. It increases the burden of blockchain and decreases the efficiency of IOPC execution. In addition to that, execution details of inner tasks are exposed on the blockchain, which leads to privacy issues. We prefer the design proposed in [13], in which each participant executes his/her own process (including the inner tasks) in his/her own BPMS. Only when a participant needs to interact with others, the BPMS interacts with the Blockchain environment, which provides a tamper-proof record for the interactive messages and business data. This design helps to not only protect the privacy of inner tasks and data, but also reduce the burden of blockchain and improve the efficiency of blockchain-based IOPC. Our smart contract generation method is designed for this context and only focuses on the interactions in multi-instance IOPC, while inner tasks not involving message exchanges are ignored.

2.2 Research on Multi-instance of Business Process

In recent years, research on multiple instances of business processes has mainly focused these topics: Process Discovery/Mining, Process Modeling and Formalising, Process Verification and Process Interaction.

Wang et al. [14] proposed an approach to discover BPMN models with sub-processes and multi-instance markers. Compared with approaches proposed by Conforti et al. [15] and Weber et al. [16], it requires less event attributes. To discover processes with multi-instance sub-processes, Liu [17] extended the Petri Net to model the processes, and developed a technique to discover these processes with event logs. Both works focus on the multi-instance within a participant process.

Corradini et al. [10] provided a formal semantics for BPMN collaborations including multiple instances of paticipants and developed an visualization tool called MIDA to help to check multi-instance collaboration models. The latest development about MIDA can be seen in [18]. Muzi et al. [19] formalised the execution semantics of service interaction patterns to remove ambiguity in BPMN collaboration. [19] was based on [10], and the formalization concerned multiple instances of tasks and processes.

Xiong et al. [11] developed a formal approach and framework to support the conformance between BPMN multi-instance choreography and collaboration. The formalization is based on the BNF syntax inspired by [10] and structured CSP# processes.

3 BPMN Collaboration Elements involving Interactions

In this section we firstly summarize the BPMN collaboration elements (as listed in Table 1) from the perspective of message interactions, then a paper review scenario will be shown as an example throughout this paper.

The element summary is inspired by [11] and [19]. Table 1 shows the summarized elements and Fig. 1 shows their notations in BPMN.

Table 1. BPMN collaboration elements related to interactions.

Element Category	Element	Abbreviation
(Ordinary) Tasks	Send Task	ST or sndTask
	Receive Task	RT or rcvTask
Multi-instance Tasks (MTs)	Parallel Multi-instance Send Task	PMST
	Parallel Multi-instance Receive Task	PMRT
	Sequential Multi-instance Send Task	SMST
	Sequential Multi-instance Receive Task	SMRT
	Time-bounded Parallel Multi-instance Send Task	TPMST
	Time-bounded Parallel Multi-instance Receive Task	TPMRT
	Time-bounded Sequential Multi-instance Send Task	TSMST
	Time-bounded Sequential Multi-instance Receive Task	TSMRT
Processes	(Ordinary) Process	P
	Parallel Multi-instance Process	PMP
Gateways	Exclusive Gateway	EG
	Parallel Gateway	PG
	EventBased Gateway	EBG

In a process (described as a "Pool" in BPMN), message exchanges are carried out through "tasks", which can be divided into Send Task (ST) and Receive Task (RT). When it comes to Multi-instance Task (MT), there are Sequential Multi-instance Send Task (SMST), Parallel Multi-instance Send Task (PMST), Sequential Multi-instance Receive Task (SMRT) and Parallel Multi-instance Receive Task (PMRT) [11]. Here, both PMST and SMST are Multi-instance Send Task (MST). In the same way, both PMRT and SMRT are Multi-instance Receive Task (MRT). So far, it is not enough to get the precise execution semantics of a MT, because it is not clear how many instances the task will have in the IOPC. So in this paper, all the MTs are assigned an attribute-*InstanceNum*.

Besides, a MT can be time-bounded (with a time-boundary event). For example, a participant wants to receive messages from others within a certain period of time. This task can be called time-bounded PMRT (TPMRT). Similarly, it is extended with new attributes - *MessageNum* and *Duration*. While *Duration* specifies the time window, and *MessageNum* means that once enough messages are received in the period, the task will stop.

Note that a process can also be multi-instance, and the instances are executed without interfering with each other. In other words, they run in parallel. So it can be called a Parallel Multi-instance Process (PMP). From the perspective of the whole IOPC, the tasks in a PMP are all MTs.

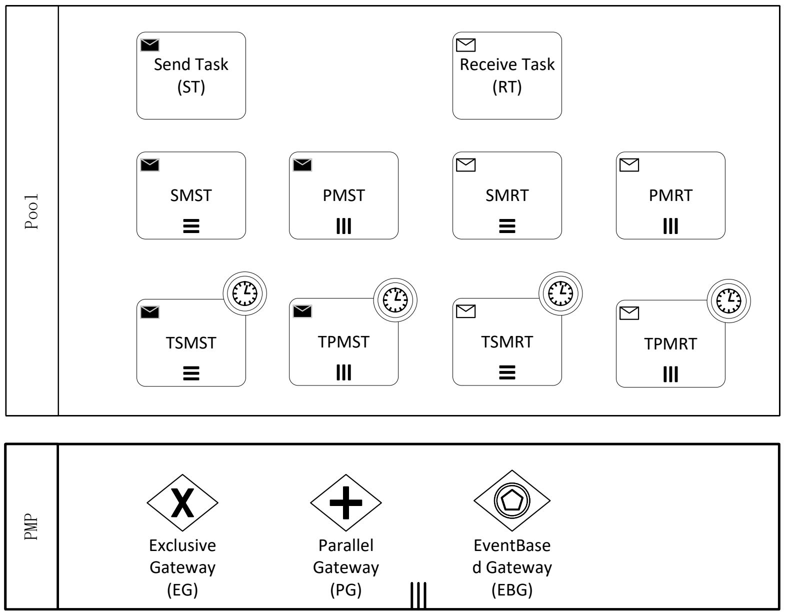

Fig. 1. BPMN notations for the elements.

In addition to the tasks and processes, gateways are also included because they affect the logic order of interactions in IOPC. Exclusive Gateway (EG) means choices. Only one of the tasks linked by a EG can be executed. While Parallel Gateway (PG) is opposite. All tasks linked by a PG can be executed at the same time. EventBased Gateway (EBG) is similar to EG, the difference is that the task to be executed depends on the occurrence of the captured external events or message reception [11].

To conclude, there are ST and RT for the ordinary tasks, PMST, SMST, TPMST and TSMST for Multi-instance Send Tasks; PMRT, SMRT, TPMRT and TSMRT for Multi-instance Receive Tasks; PMP for Multi-instance Processes. EG, PG and EBG for Gateways.

Fig. 2 shows an IOPC scenario adapted from [10]. It focuses on the interactions and removes the inner tasks of the participants. In this collaboration, *Program Committee Chair* (short for *Chair*) assigns a paper to more than one (e.g., 5) *Reviewers*, who are required to review the paper and submit reviews in a certain period of time (e.g., a month). If a specified number (e.g., 3) of reviews are received, *Chair* stops receiving reviews, makes evaluation and decides to accept or reject the paper. The result will be sent to the (*Contact*) *Author*, and relevant *Reviewers* will get the feedback. If submitted reviews are not enough, this paper reviewing fails. *Chair* will send the result and feedback to inform the *Reviewers* and *Author* that there are not enough reviews to evaluate the paper.

The task *Assign Paper* is a SMST because the paper will be sent to multiple reviewers sequentially, so is *Send Feedback*. *Receive Reviews* is a time-bounded

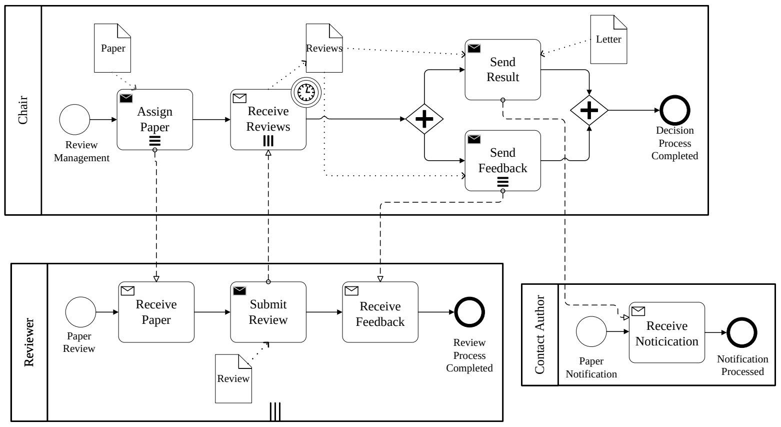

Fig. 2. BPMN collaboration diagram of paper review scenario.

PMRT. It means that *Chair* wants to receive *MessageNum* reviews within *Duration*. The process of *Reviewer* is modelled as a multi-instance pool, that is a PMP. Because more than one reviewer do the paper review work in the same period. While *Author* is an ordinary process and the only interaction is to receive notification from *Chair* (except for submitting the paper in the early period).

It is possible for this IOPC to encounter a deadlock. For example, *Chair* wants to receive 3 reviews, but the *InstanceNum* of *Reviewer* is set to 2 or *Submit Review* is marked as a Receive Task.

To implement the collaboration above in blockchain, the collaboration model needs to be translated to a smart contract which contains precise semantics. Besides, to avoid the loss caused by deployment of a flawed model, it is necessary to verify the model in advance.

4 Smart contract Generation from BPMN to Solidity

This section details our smart contract generation method. There are three phases to generate smart contracts. First, it transforms a BPMN collaboration model to CSP#. Then it verifies the soundness of the CSP# model with an available analysis tool. Finally, it generates smart contract code for every task. To achieve the goals:

- We give a CSP# formalization for the BPMN collaboration elements related to interactions, and a transformation algorithm so that we can transform the collaboration models to CSP# models.
- We define the soundness of the CSP# models in this context so that we can verify them with available CSP# analysis tools.
- We develop a set of algorithms to parse the CSP# processes based on the syntax trees and generate smart contract code for each atomic CSP# process (corresponding to a task in IOPC).

4.1 Transformation form BPMN to CSP# Model

[11] has provided a CSP# formalization for the MTs without time boundaries. In this paper, we adapt it to support the time-bounded MTs and PMP. Table 2 shows a part of our formalization rules. Formalization for PMP is not shown but can be represented based on Table 2 because a process is composed of tasks and gateways. In Table 2, M refers to the message flow that can be describe as (sender, receiver, m), where m is a message; And n refers to the instance number of the MT. The feature of the formalization is that every task for message interactions is described as a csp# process sending a message through a channel (e.g. $ch!m$, where "ch" means a message channel, and "!" means sending) or receiving a message from a channel (e.g. $ch?m$, where "?" means receiving). And for MTs without time boundaries, they are described as csp# processes running in parallel (e.g., |||i:{1..n}@(sndTask), where "|||i:{1..n}" means that the n csp# processes run in parallel) or sequentially (e.g., if(i<n){rcvTask; SMRT(i+1)}).

Table 2. A CSP# formalization for core collaboration model elements.

Elements	BPMN Syntax	CSP#
Tasks	sndTask(M)	ch!m ->Skip;
	rcvTask(M)	ch?m ->Skip;
	PMST(sndTask,n)	|||i:{1..n}@(sndTask)
	PMRT(rcvTask,n)	|||i:{1..n}@(rcvTask)
	SMST(sndTask,n)	if(i<n){sndTask; SMST(i+1)}
	SMRT(rcvTask,n)	if(i<n){rcvTask; SMRT(i+1)}
	TPMST(PMST,msgNum)	pcase{PMST(msgNum)}
	TPMRT(PMRT,msgNum)	pcase{PMRT(msgNum)}
	TSMST(SMST,msgNum)	pcase{SMST(msgNum)}
	TSMRT(SMRT,msgNum)	pcase{SMRT(msgNum)}
Gateways	EG(T1,T2...Tn)	(T1 [] T2 []. . . [] Tn);
	PG(T1,T2...Tn)	(T1 || T2 || ... || Tn);
	EBG(T1,T2...Tn)	(T1 [*] T2 [*]. . . [*] Tn);

Time-bounded MTs are special, we use Probability CSP# (PCSP#) [20] to formalize them. Initially, for time-bounded MTs, we use CSP# with time-extension to describe the semantics. It can describe the semantics well. However, it will take a lot of time to verify the CSP# model. In our early trial, it took more than 30 minutes to verify a model composed of 24 atomic CSP# processes, only two of which are assigned with time attributes between 1-5. If we remove the time attributes, model verification can be finished in 0.5 seconds. So we find the alternative - PCSP#, which supports probability assignment for the CSP# processes. For time-bounded MTs, the number of task instances executed within the specific period (*Duration*) is indeterminate (randomly between 1 and *MessageNum*). With PCSP#, we can remove the time attribute *Duration*, describe all the cases and assign each case an execution probability so that model verification can be completed quickly. And when generating smart contract, we recover the attribute and generate code according its original semantics.

Take *Receive Reviews* in Fig. 2 as an example. There are 3 possible execution results for this TPMRT: *Chair* receives 1, 2 or 3 reviews in a month. Each case can be taken as a PMRT. So this TPMRT can be taken as a composition of 3 PMRTs with execution probabilities, which can be described with PCSP#.

According to the formalization, we can get the CSP# processes for every participant in Fig. 2, as shown in Code 1.1. In Code 1.1, *Chair* is a composite CSP# process (defined in line 13) that contains four CSP# processes (defined in line 1-13), while *Reviewer* (defined in 15-16) contains two.

Code 1.1. CSP# processes for Paper Review collaboration.

```
1  AssignPaper(i) = if (i< 5){    //SMST(AssignPaper,InsNum=5)
2       PaperChannel!AssignPaper−>Skip;
3       AssignPaper(i+1)};
4  ReceiveReviews() = pcase { //pcase{PMST(MsgNum=3)}
5           1:  ReviewChannel?ReceiveReviews−>Skip
6           1:  ||| i :{0..1} @ReviewChannel?ReceiveReviews−>Skip
7           1:  ||| i :{0..2} @ReviewChannel?ReceiveReviews−>Skip
8  };
9  SendFeedback(i) = if (i< 5){  //SMST(SendFeedback,InsNum=5)
10      FeedbackChannel!Feedback−>Skip;
11      SendFeedback(i+1)};
12
13 Chair() = AssignPaper(0);ReceiveReviews();(ResultChannel!SendResult−>Skip ||
         SendFeedBack(0));
14 //PMP(process, InsNum=5)
15 Reviewer() = ||| i :{0..4} @(
16 PaperChannel?AssignPaper−>Skip; ReviewChannel!ReceiveReviews−>Skip);
```

To support the automatic transformation from BPMN collaboration model to CSP#, we develop a transformation algorithm, part of which is shown in Algorithm 1. It first parses the BPMN collaboration model (line 1), then generates csp# code for message definitions and channel definitions (line 2-4). Finally, it generates csp# code for each participant process according to Table 2 (line 8-21).

4.2 Model Verification

We verify the collaboration model by verifying the corresponding CSP# model. In this paper, we mainly verify the soundness because it is fundamental for IOPC.

A multi-instance IOPC is sound if all participant processes are terminable, deadlock-free and all the tasks are reachable. The formal definition is:

$\forall$ *participant* $\in P$: deadlockfree(*participant*) $\wedge$ <> ($\forall$ *participant* $\in P$: *participant* reaches *end*)

where *deadlockfree*, *nonterminating* and *reaches* are CSP# based attribute assertions, *end* is a given conditional proposition.

With the definition, it is easy to check the soundness of the CSP model with existing CSP# analysis tools like PAT [12].

4.3 Contract Generation from CSP# Model to Solidity

When the CSP# model passes the soundness verification, it is time to generate the smart contract. This section is inspired by the knowledge of lexical and

Algorithm 1: A part of Transfomation Algorithm

```
Input: BPMN collaboration model in XML
Output: CSPCode
1  BPMNodes, MsgFlow, SeqFlow = ParseBPMNModel(); // parsing the BPMN
      collaboration model, getting the elements related to interactions
      (stored in BPMNodes), MessageFlow (stored in MsgFlow) and Sequence Flow
      (Stored in SeqFlow)
2  for msgflow in MsgFlow do // generating the code that defines messages and
    channels
       // generating the code that defines a message for msgflow
3      MsgCode += CodeForMsgDef(msgflow);
       // generating the code that defines a message channel for msgflow
4      ChCode += CodeForChDef( msgflow);
5  CSPCode += MsgCode;
6  CSPCode += ChCode;
   // get the id for each participant process
7  participantProcesses = getParticipantProcessID(BPMNodes);
8  for p in participantProcesses do
       // generating the csp# code for each participant
9      element = getStartEvent(p, BPMNodes); // get the start event for p
10     while element != null do
11         if element.type is startEvent then
12             element = SeqFlow[element];
13             continue;
14         if element.type is one of the TaskElements then
               // generate code for the task according to Table 2
15             ProcCode += CodeForTask(element);
16             element = SeqFlow[element];
17             continue;
18         if element.type is one of the Gateways then
               // generate code for the tasks in the gateway according to
                  Table 2
19             element, ProcCode = CodeForTasksInGateway(element, ProcCode);
20             continue;
21     CSPCode += ProcCode;
22 return CSPCode;
```

syntax analysis. It is based on the syntax trees and takes three steps to get the final contract.

Step 1: Preliminary Association Relationships Extraction. First, we abstract the interaction logic of the CSP# processes. This is based on the syntax trees of the CSP# processes. The syntax trees are obtained with the help of antlr4 [21], which helps us to get precise semantics of the CSP# process. Fig. 3 shows an example. In the syntax tree, leaf nodes form the CSP processes, and the content of a non-leaf-node is the composition of all the related leaf-nodes, which is always a composite CSP# process.

We define 6 association relationships to describe the interaction logic. They are *Init*, *End*, *Next*, *And*, *Xor* and *Enable*:

- *Init* and *End* describe the relationships between a CSP# parent process and some (but not all) of its child processes, where a parent (child) process is

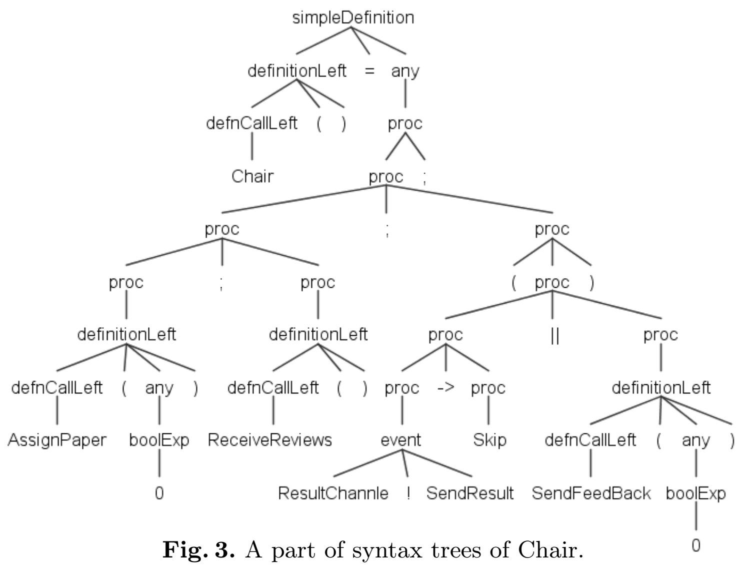

Fig. 3. A part of syntax trees of Chair.

a CSP# process represented by a parent (child) node in the syntax tree. *Init*[P]=C[] means that if process P begins, then it will begin the processes in C[]. And *End*[C]=P means that if process C ends its execution, then it will tell P to end its execution.

- *Next* describes the sequential logic between the CSP# processes within a participant. *Next*[A]=B[] means that if process A ends its execution, then processes in B[] can start its execution.
- *And* describes the relationship among the CSP# processes in a parallel gateway. *And*[A]=B[] means that process A and processes in B[] can be executed in parallel, and only when all of them completes, the task after the parallel gateway can start.
- *Xor* describes the relationship among the CSP# processes in an exclusive gateway. *Xor*[A]=B[] means that if process A starts its execution, then all the processes in B[] can not start any more.
- *Enable* describes the relationship related to message interactions. If A is a CSP# process to send a message, and B is a CSP# process to receive the message, then *Enable*[A]=B. It means that only when a message has be sent, it can be received.

Then we develop a traversal algorithm to capture the association relationships with the syntax trees. Algorithm 2 shows a part of the algorithm. It traverses the syntax trees and records the relationships according to the content of the nodes. Besides, it supports linking a subtree to another syntax tree. This makes description of CSP# more flexible and convenient.

Fig. 4 shows a part of relationships of participant *Chair*, which involves more than one syntax tree. It can be observed that it contains too many intermediate

Algorithm 2: A part of Relationship Traversal Algorithm

```
Input: Syntax tree node with subtrees
// Traverse the node with (3 or more) subtrees
// Node.LeftS: process composed of leaf nodes of the left subtree
// Node.RightS: process composed of leaf nodes of the right subtree
// Name: identifier of a defined process
// Def: ‘‘definitionLeft’’ of a subtree
if VisitEqual (Node.MiddleS) then // "="operator, meet process definition
    add Node.RightS to Init[Node.LeftS.Name];
    add Node.LeftS.Name to Node[Node.RightS];
if VisitDef (Node) then // meet the left part of a process definition
    add Node.Name to Init[Node];
    add Node to Node[Node.Name];
if VisitPME(Node) then // meet a parallel multi-instance task or process
    get InsNum, MsgNum (if exists) and TaskContent;
    add TaskContent to Init[Node];
    add Node to End[TaskContent];
if VisitSME(Node) then // meet a sequential multi-instance task or process
    get InsNum, MsgNum(if exists) and TaskContent;
    add TaskContent to Init[Node];
    add Node to End[TaskContent];
if VisitTME(Node) then // meet a time-bounded multi-instance task
    get InsNum, MsgNum(if exists),Duration and TaskContent;
    add TaskContent to Init[Node];
    add Node to End[TaskContent];
return Next, End, Init, And, Xor;
```

composite CSP# processes to be suitable as input for smart contract generation. So simplification is needed.

Step 2: Reduced Association Relationship Extraction. The relationships in Step 1 are defined to capture the logical order of each process in the syntax trees so that there is no missing part. However, due to the non-leaf nodes, corresponding to the CSP# composite processes, the state transitions are too complex and redundant to make the smart contract generated acceptable. So we take some measures to reduce them. The principle of reduction is to focus on the leaf nodes, that is the **atomic** CSP# processes like *ResultChannel*!*SendResult* ->*Skip*

Here we define three new relationships to achieve our reduction: *Activate*, *Inactivate*, *Parallel*. They are similar to the *Next*, *Xor* and *And* in Step 1. The difference is that they only focus on the atomic csp# processes. We develop a Reduction Algorithm, as shown in Algorithm 3, to get the new relationships based on the relationships in step 1. It removes the redundant state transitions caused by the non-leaf nodes. Finally, *Activate*, *Inactivate*, *Parallel* and *Enable* are enough to describe the logic order of interactions.

Fig. 5 shows the reduced relationships of participant *Chair*. It includes atomic CSP# processes on multiple syntax trees related to the participant and removes all the composite processes, making state transitions simpler. Each multi-instance task in Fig. 5 is annotated with its type and the extended attributes.

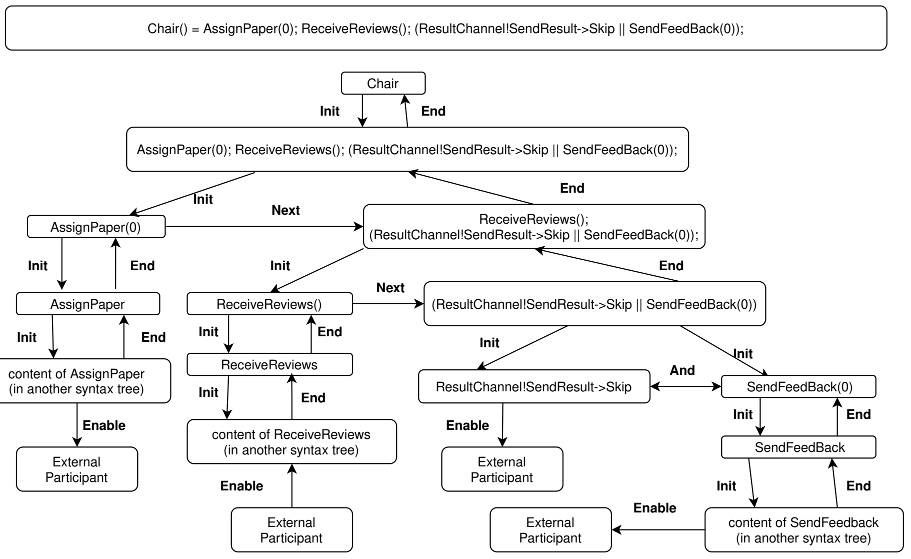
Chair() = AssignPaper(0); ReceiveReviews(); (ResultChannel!SendResult->Skip || SendFeedBack(0));
Chair
Init
End
AssignPaper(0); ReceiveReviews(); (ResultChannel!SendResult->Skip || SendFeedBack(0));
Init
Next
End
AssignPaper(0)
ReceiveReviews(); (ResultChannel!SendResult->Skip || SendFeedBack(0));
AssignPaper
content of AssignPaper (in another syntax tree)
Enable
External Participant
ReceiveReviews()
ReceiveReviews
content of ReceiveReviews (in another syntax tree)
External Participant
(ResultChannel!SendResult->Skip || SendFeedBack(0))
ResultChannel!SendResult->Skip
And
SendFeedBack(0)
SendFeedBack
content of SendFeedback (in another syntax tree)
External Participant
External Participant

Fig. 4. A part of the relationships of *Chair*.

Step 3: Smart Contract Generation. After the processing of step 1 and step 2, our smart contract generation method generates functions for every atomic CSP# process (that is every interaction task in the collaboration model) with a template. The functions form the main content of the contract and control the execution of the collaborative process in the specific order, which we capture above through *Activate*, *Inactivate*, *Parallel* and *Enable*. These functions are generated to handle external requests. With different templates, we can generate different code for these functions, which makes it convenient to customize the smart contracts. Code 1.2 in Appendix shows an example of contract code generated with our method.

Algorithm 4 shows a brief processing logic for a function related to a sequential multi-instance task. It first checks whether the task is waiting for execution (line 1-2) and whether it meets the requirements to execute (line 3, line 4 and line 9). Next is to behave according to the type of the task (line 5-8, line 10-11), such as sending a message, increasing the *count* that indicates how many instances the MT has now. When the MT has its first instance, it will prevent the corresponding tasks in *Inactivate* to execute (line 12-14). If the MT has enough instances (line 15), it will complete its execution (line 16) and makes the next tasks (stored in *Activate*) waiting to be executed (line 17-23).

For ordinary tasks (that is Send Tasks and Receive Tasks), The processing is similar and simpler: sending or receiving a message according to the type of the task when the the requirements to execute are met, preventing the corresponding tasks in *Inactivate*[] from being executed and making the tasks in *Activate*[] wait to be executed.

For parallel multi-instance tasks, the processing is also similar. The key is to control the synchronization of *count*. And for a PMP, every instance of it

Algorithm 3: A part of Relationship Reduction Algorithm

```
   Input: CSP# procs of the participants and Relationships: Init, End, Next, And, Xor
   // get Activate
1  for ap in AtomicProcesses do
       // find a composite csp# proc nxtCP with End[] and Next[] (if ap
          ends, then nxtCP is the next composite proc to start. It looks like
          searching up (with the help of End[])in the syntax tree)
2      nxtCP=FindNextCompositeProc(ap);
3      for child in Init[nxtCP] do
           // find atomic processes in Init[nxtCP]. It looks like searching
              down in the syntax tree
4          nxtAP = FindAtomicProcesses(child); // with the help of Init[]
5          add nxtAP to Activate[ap];

6  for xorp in ExclusiveGatewayProcesses do // xorp is a composite csp# proc
     that contains all atomic procs in a EG. Here we assume there are two
     outgoing paths
       // find the first atomic processes in every outgoing path
       // with the help of Init[]
7      FirstFAPGroup=FindFisrtAtomicProcesses(Xor[xorp][1]);
8      SecondFAPGroup=FindFisrtAtomicProcesses(Xor[xorp][2]);
9      for ap in FirstAPGroup do
10         add SecondAPGroup to Inactivate[ap];
11     for ap in SecondAPGroup do
12         add FirstAPGroup to Inactivate[ap];

13 for andp in ParallelGatewayProcesses do
       // find the last atomic processes in every outgoing path
       // with the help of Init[] and Next[]
14     FirstLAPGroup=FindLastAtomicProcesses(And[andp][1]);
15     SecondLAPGroup=FindLastAtomicProcesses(And[andp][2]);
16     for ap in FirstLAPGroup do
17         add SecondLAPGroup to Inactivate[ap];
18     for ap in SecondLAPGroup do
19         add FirstLAPGroup to Inactivate[ap];

20 return Activate,Inactivate, Parallel;
```

belongs to different participants. Assigned unique ids, the PMP instances can be taken as different ordinary processes. So the tasks of each PMP instance can be processed as ordinary tasks in ordinary processes.

5 Evaluation

This section we evaluate our smart contract generation method. First, we compare the features of our method with ones of related work. Second, we setup an experiment to test the model verification ability of our method and the cost of the generated contracts.

5.1 Comparison with Related Method

Table 3 shows the features of our method and methods in literature [4], [8] and [2]. The major contribution of our work is that we support all multi-instance elements introduced in Section 3, while others don't support any of them.

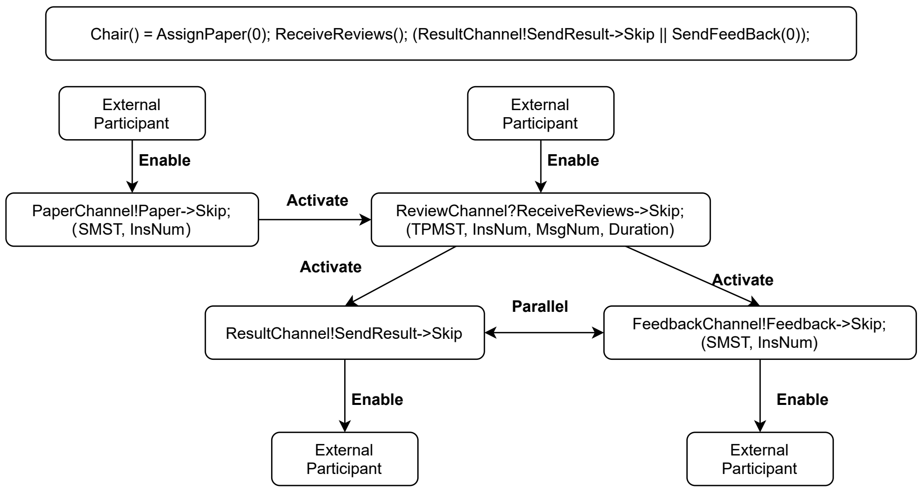
Chair() = AssignPaper(0); ReceiveReviews(); (ResultChannel!SendResult->Skip || SendFeedBack(0));
External Participant
Enable
External Participant
Enable
PaperChannel!Paper->Skip; (SMST, InsNum)
Activate
ReviewChannel?ReceiveReviews->Skip; (TPMST, InsNum, MsgNum, Duration)
Activate
Activate
Parallel
ResultChannel!SendResult->Skip
FeedbackChannel!Feedback->Skip; (SMST, InsNum)
Enable
Enable
External Participant
External Participant

Fig. 5. Reduced relationships of *Chair*.

Method in [4] is a classic one in smart contract generation. It supports and focuses the interactions among the participants of IOPC. But it doesn't include any model verification measures, which makes the generated smart contract possibly flawed, resulting in loss for unexpected process execution.

[8] and [2] use similar steps to generate smart contracts and have the same goal - optimize the cost of smart contract. Both of them translate the process model to a formal intermediate. For [8], it is Petri Net. And for [2], it is state chart. This makes model verification possible. However, none of BPMN process model, Petri Net or state chart can provide support for description and verification of interactions in IOPC without any extension, which is an important feature and reflects how participants cooperate to achieve the collaboration. And if they are extended, extra work for properties validation and tool development is needed. Our choice of BPMN collaboration model and CSP# helps us avoid this. It is convenient to abstract interaction information from a collaboration diagram and describe the interactions with CSP#. Because there is no additional property extension, model verification can be done with existing analysis tools.

To conclude, our method supports all the multi-instance elements mentioned, description of interactions in IOPC, and model verification from interactions perspective.

5.2 Experiment

We collect 4 cases of multi-instance IOPC: Paper Review (PR), Order Fulfilment (OF), Cake Preparation (CP) and Supply Chain (SC) from [10], [19], [18] and [11] respectively. These cases are adapted so that they can pass our model verification and cover all of the multi-instance elements without time boundaries and a part of time-bounded elements. It is acceptable. Because in this experiment, the cost

Algorithm 4: A part of ExternalRequest Algorithm

```
Input: CSP# atomic proc
if State(proc) is not Waiting then
    return false;
if proc is (Time-bound) SMT (and timestamp < proc.ddl) then // proc is (T)SMST or (T)SMRT
    if proc is SMST then
        emit message event of rcvProc; // rcvProc = Enable[proc[count]];
        set RcvEnabled[rcvProc] as true; // the receive condition is triggered
        stateChange(proc, Executing); // execute the task
        proc.count ++; // count refers to the number of the task instances
    if proc is SMRT and RcvEnabled[proc] == true then // execute the Receive Task
        stateChange(proc, Executing);
        proc.count ++;
if count == 1 and Inactivate[proc] exists then // at first call of this function
    for xorProc in Inactivate[proc] do // inactivate the xor procs
        stateChange(xorProc, Disabled);
if (count == InsNum) or (proc is time-bounded and count == proc.MsgNum) then // enough instances
    stateChange(proc, Done); // No more instances for this task if Activate[proc] exists then // Activate the next procs
        if Parallel[proc] exists and one of them is not Done then
            do nothing;
        else // next procs are permitted to be executed
            for nextProc in Activate[proc] do
                stateChange(nextProc, Waiting);
                if nextProc is time-bounded then
                    nextProc.ddl = (timestamp + nextProc.Duration);
else // instances not enough
    stateChange(proc, Waiting); // More instances of this task are permitted
```

depends on the contract content and the number of executions of each function. For comparison with other methods, the only acceptable variable is the contract content. This makes time boundaries not important.

Model Verification Before cost evaluation, to test the model verification ability of our method, 5 copies of each case are made and modified to cause a control flow deadlock. These modifications can be changes in task types or process types, changes in attribute values, or deletions of elements. For example, in Fig. 2, if *Chair* wants to receive 3 reviews, we can set the *InstanceNum* of *Reviewer* to 2 or change the type of *Submit Review* to Receive Task to cause a deadlock. Then these flawed models are transformed to CSP# models according to Table 2 and verified. Unsurprisingly, none of them passes the verification.

Cost Evaluation Only support for interaction description and model verification is not convincing enough. The contracts are used to executed in blockchain, and the cost is an important indicator for users.

We generate smart contracts for the four cases with our method (labelled as A) and test the gas cost to complete the IOPC in the environment shown

Table 3. Features Comparison with Related Methods.

Features	**Our method**	**[4]**	**[8]**	**[2]**
Begin with	BPMN collaboration model	BPMN choreography model	BPMN process model	BPMN process model (with lanes)
Interaction description	Yes	Yes	No	No
Formal intermediate	CSP#	No	Petri Net	State Chart
Model Verification Included	Yes	No	No	No
Cost Optimization	No	No	Yes	Yes
Multi-instance Support	All elements mentioned	None (but can be extended)	none	none

in Table 4. There are two comparison groups. One (labelled as B) generates contracts with the method extended from [4] . The other (labelled as C) generates contracts manually based on the idea of [2]. The instance numbers of each multi-instance elements in each case are consistent in three groups. Table 5 shows the result.

Table 4. Experiment Configurations.

Item	**Configurations**
Machine	8 cores, with AMD Ryzen 7 4800H@2.9GHz and 8GB RAM
Operating System	Ubuntu 20.04
Blockchain Enviroment	Ganache (an Ethereum simulator), with 10 nodes
Contract Compiler	Solc 0.8.19

Table 5. Gas Cost Comparison

Case	**Cost Item**	**our method (A)**	**B**	**C**
PR	Total Cost	1382242	1417043	1039798
	Compared with C	133%	136%	100%
OF	Total Cost	1300899	1147732	1037978
	Compared with C	125%	111%	100%
CP	Total Cost	1701442	1459754	1268307
	Compared with C	134%	115%	100%
SC	Total Cost	2724285	2502170	1929297
	Compared with C	142%	130%	100%

In general, our method has the highest cost, while C has the lowest cost. It is within our expectation. As automatic methods, A and B need to handle more variables and state transitions than C, leading to more gas cost for contract initialization and execution. And compared with B, our method consumes (12%-19%) more gas, especially when there are many multi-instance elements. This is the shortcoming of our method because of our lack of optimization. However, our method provides model verification before contract generation to avoid the unexpected implementation and execution of IOPC, which always costs more. In this context, 12%-19% of extra gas cost is acceptable.

6 Conclusion

This paper proposes a method to support model verification and smart contract generation for multi-instance IOPC. We focus on the elements related to interactions in multi-instance IOPC. Thus we make use of CSP#, which supports formal description of interactions well, to formalize the semantics of BPMN collaboration, making model verification feasible. After the formalization, we generate the smart contract based on the syntax trees of CSP# processes. Compared with other research, our method supports all of the multi-instance elements we have mentioned, and model verification from interactions perspective. Both of the features are important. One expands the applicability and utility of blockchain-based IOPC. The other prevents models with flaws from being translated into smart contracts, avoiding the loss of unexpected execution of IOPC.

As the experiment shows, our method costs more gas. In the future, we will optimize the generation to reduce the cost. Besides, it is planned to expand the CSP# formalization to support more elements like inclusive gateway. Also, support for user-defined properties verification for IOPC with CSP# is another future goal.

Acknowledgments. This work is supported by the NSFC-Guangdong Joint Fund Project under Grant Nos. U1911205, U20A6003; the National Natural Science Foundation of China (NSFC) under Grant No. 61972427; the Research Foundation of Science and Technology Plan Project in Guangdong Province under Grant No. 2020A0505100030.

References

1. Van der Aalst, W.: Loosely coupled interorganizational workflows: modeling and analyzing workflows crossing organizational boundaries. Inf. Manage. **37**(2), 67–75 (2000)
2. Nakamura, H., Miyamoto, K., Kudo, M.: Inter-organizational Business Processes Managed by Blockchain. In: Hacid, H., Cellary, W., Wang, H., Paik, HY., Zhou, R. (eds) WISE 2018. LNCS, vol 11233. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02922-7_1
3. Mendling, J., Weber, I., Aalst, W.V.D., et al.: Blockchains for business process management-challenges and opportunities. ACM Trans. Manag. Inf. Syst. **9**(1), 1–16 (2018)
4. Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A., Mendling, J.: Untrusted Business Process Monitoring and Execution Using Blockchain. In: La Rosa, M., Loos, P., Pastor, O. (eds) BPM 2016. LNCS, vol 9850, pp. 329–347. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45348-4_19
5. Lòpez-Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I., Ponomarev, A.: Caterpillar: a business process execution engine on the Ethereum blockchain. Softw. Pract. Exp. **49**(7), 1162–1193 (2019)
6. Tran, A.B., Lu, Q., Weber, I.: Lorikeet: a model-driven engineering tool for blockchain-based business process execution and asset management. In: BPM (Dissertation/Demos/Industry), pp. 56–60 (2018)

7. Corradini, F., Marcelletti, A., Morichetta, A., Polini, A., Re, B., Tiezzi, F.: Engineering trustable choreography-based systems using blockchain. In: Hung, C., Cerný, T., Shin, D., Bechini, A. (eds.) SAC2020, pp. 1470–1479. ACM (2020). https://doi.org/10.1145/3341105.3373988
8. García-Bañuelos, L., Ponomarev, A., Dumas, M., Weber, I.: Optimized execution of business processes on blockchain. In: Carmona, J., Engels, G., Kumar, A. (eds.) BPM 2017. LNCS, vol. 10445, pp. 130–146. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65000-5_8
9. Tonga Naha, R., Zhang, K.: Pupa: Smart Contracts for BPMN with Time-Dependent Events and Inclusive Gateways. In: , et al. Business Process Management: Blockchain, Robotic Process Automation, and Central and Eastern Europe Forum. BPM 2022. LNBIP, vol 459. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-16168-1_2
10. Corradini, F., Muzi, C., Re, B., Rossi, L., Tiezzi, F.: Animating multiple instances in BPMN collaborations: from formal semantics to tool support. In: Weske, M., Montali, M., Weber, I., vom Brocke, J. (eds.) BPM 2018. LNCS, vol. 11080, pp. 83–101. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98648-7_6
11. Xiong, T., Pan, M., Yu, Y., Lou, D.: Conformance between Choreography and Collaboration in bpmn involving multi-instance participants. Int. J. Pattern Recognit. Artif. Intell. **36**(07), 2259013 (2022)
12. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Bouajjani, A., Maler, O. (eds) CAV 2009. LNCS, vol 5643, pp. 709–714. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_59
13. Tang, X., Yu, Y., Wu, J., Pan, M.: Busines Process Interoperability Service Framework based on blockchain. Computer Integrated Manufacturing Systems **27**(9), 2508 (2021)
14. Wang, Y., Wen, L., Yan, Z., Sun, B., Wang, J.: Discovering BPMN Models with Sub-processes and Multi-instance Markers. In: , *et al.* OTM 2015. LNCS, vol 9415, pp. 185–201. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26148-5_11
15. Conforti, R., Dumas, M., García-Bañuelos, L., La Rosa, M.: Beyond tasks and gateways: discovering BPMN models with subprocesses, boundary events and activity markers. In: Sadiq, S., Soffer, P., Völzer, H. (eds.) BPM 2014. LNCS, vol. 8659, pp. 101–117. Springer, Cham (2014). https://doi.org/978-3-319-10172-9_7
16. Weber, I., Farshchi, M., Mendling, J., Schneider, J.: Mining processes with multi-instantiation. In: SAC2015, pp. 1231–1237. ACM (2015). https://doi.org/10.1145/2695664.2699493
17. Liu, C.: Formal modeling and discovery of multi-instance business processes: A cloud resource management case study. IEEE-CAA J. Automatica Sin. **9**(12), 2151–2160 (2022)
18. Corradini, F., Muzi, C., Re, B., Rossi, L., Tiezzi, F.: Formalising and animating multiple instances in bpmn collaborations. Inf. Syst. **103**, 101459 (2022)
19. Muzi, C., Pufahl, L., Rossi, L., Weske, M., Tiezzi, F. (2018). Formalising BPMN Service Interaction Patterns. In: Buchmann, R., Karagiannis, D., Kirikova, M. (eds) PoEM 2018. LNBIP, vol 335. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02302-7_1
20. Morgan, C., McIver, A., Seidel, K. et al. Refinement-oriented probability for CSP. Form. Asp. Comput. **8**, 617–647 (1996). https://doi.org/10.1007/BF01213492

21. Parr, T.J., Quong, R.W.: ANTLR: A Predicated- LL(k) Parser Generator. Softw. Pract. Exp. **25**(7) 789–810 (1995).

7 Appendix

Here we show an example of smart contract code generated for the Paper Review collaboration.

Code 1.2. Part of contract code for Paper Review collaboration.

```
function AssignPaper() external {  // a SMST
    if (isActiveAssignPaper) { // check the state
        emit mesag("", "", "AssignPaper"); // sending msg to blockchain
        isEnabledReceivePaper++; // enable the receive task
        CountAssignPaper ++; // increase the instance count
        if (CountAssignPaper == 5){ // enough instances
            isActiveAssignPaper = false;  // no more insatnce for this task
            isActiveReceiveReviews = true;// next task can start
            // time window for next task
            DDLReceiveReviews = block.timestamp + DurationReceiveReviews;
        }
    }
}
function ReceivePaper() external{  // a task in a PMP
    if ((isActiveReceivePaper) && (isEnabledReceivePaper > 0)) { // check the state
        emit mesag("", "", "ReceivePaper"); // sending msg to blockchain
        CountReceivePaper ++; // increase the instance count
        isEnabledReceivePaper --;
        isActivatedSubmitReview ++; // next task can have a instance
        if (CountAssignPaper == 5){ // enough instances
            isActiveReceivePaper = false;   // no more insatnce for this task
        }
    }
}
function ReceiveReviews() external{ // a TPMRT
    // check the state and time
    if (isActiveReceiveReviews && (block.timestamp < DDLReceiveReviews)) {
        emit mesag("", "", "ReceiveReviews"); // sending msg to blockchain
        CountReceiveReviews ++; // increase the instance count
        if (CountReceiveReviews == 3){ // enough instances
            isActiveReceiveReviews = false;  // no more insatnce for this task
            isActiveSendResult = true;  // next tasks can start
            isActiveteSendFeedback = true;
        }
    }
}
```