Controlled Flexibility in Blockchain-Based Collaborative Business Processes

Orlenys López-Pintado[a,*], Marlon Dumas[a], Luciano García-Bañuelos[b], Ingo Weber[c]

[a] *University of Tartu, Estonia*
[b] *Tecnológico de Monterrey, Mexico*
[c] *Technische Universität Berlin, Germany*

Abstract

Blockchain technology enables the execution of collaborative business processes involving mutually untrusted parties. Existing tools allow such processes to be modeled using high-level notations and compiled into smart contracts that can be deployed on blockchain platforms. However, these tools do not provide mechanisms to cope with the flexibility requirements inherent to open and dynamic collaboration environments. In particular, existing tools adopt a static role binding approach wherein roles are bound to actors upfront when a process instance is created. Also, these tools do not allow participants to collectively make choices regarding alternative sub-processes or branches in the process model, at runtime. This paper presents a model for dynamic binding of actors to roles in collaborative processes and an associated binding policy specification language. The proposed language is endowed with a Petri net semantics, thus enabling policy consistency verification. Furthermore, the paper introduces a model for consensus-based control-flow flexibility, wherein participants in a process can collectively agree on how to steer the business process within the boundaries defined by control-flow agreement policies. The paper also outlines an approach to compile policy specifications into smart contracts for enforcement. An experimental evaluation shows that the cost of policy enforcement increases linearly with the number of roles, control-flow elements, and policy constraints.

Keywords: Collaborative Business Process, Blockchain, Flexibility

[*]Corresponding author
Email address: `orlenyslp@ut.ee` (Orlenys López-Pintado)

1. Introduction

Executing collaborative inter-organizational business processes, particularly in open environments where the set of participants is not fixed in advance, is a long-standing challenge. This challenge arises from a fundamental tension between, on the one hand, the need to allow some flexibility in the execution of the process so as to accommodate a diverse range of requirements and, on the other hand, the lack of mutual trust between participants [1].

In recent years, blockchain technology has emerged as an appealing medium for collaborative business process execution between mutually untrusted parties [2]. Several approaches have been proposed to execute and monitor collaborative processes using high-level process specifications, such as process models captured in the Business Process Model and Notation (BPMN) [3, 4, 5, 6]. These approaches, however, suffer from two limitations, which hinder their applicability in dynamic environments:

1. They either do not provide a mechanism to bind actors to roles or, when they do, they do not allow actors to be bound to roles dynamically.

2. They assume that the schema of the business process is fixed, and do not provide flexibility mechanisms to enable actors to steer a process instance in order to fit their collective requirements.

To illustrate the need for the first of the above flexibility features, we consider a business-to-business purchasing process involving a buyer, a supplier and a carrier. A buyer may trust a given carrier but not others, even though they all play the same role. Additionally, trust relations may change dynamically. For example, a buyer may initially trust a carrier and agree to its appointment with the endorsement of the supplier. But later, the buyer may lose this trust (e.g. if the carrier misses a deadline). Thereafter, the buyer may wish to re-bind the transportation task to another carrier in consultation with the supplier. This example illustrates the need to support *dynamic binding and un-binding of actors to roles* and *collaborative decision making about role bindings* (buyer and supplier both need to agree on the carrier).

Meanwhile, to illustrate the need for the second flexibility feature, we consider the case where some buyers require that the carrier uses a specific type of customs declaration system. Other buyers however, require a different system. The choice of customs declaration system depends on the carrier's and the supplier's capabilities and constraints, which vary from one carrier or supplier to another. Hence, the decision on which system to use, needs to be made dynamically by common agreement between the buyer, the seller, and the carrier. This example illustrates the *need for participants to collectively make choices regarding alternative sub-processes or branches in the process model, in an environment where new participants may join, and the capabilities and constraints of the participants may change over time.*

The above examples illustrate the need for flexible execution mechanisms in collaborative processes. These flexibility mechanisms, however, need to be designed in a way that takes into account the lack of mutual trust between participants. In other words, flexibility in such environments needs to be accompanied by control mechanisms allowing the participants to collectively decide on the course of execution of each process instance. In the above example, it should not be possible for the buyer alone to appoint a carrier, as this appointment also affects the supplier. Similarly, it should not be possible for the buyer alone to decide on the customs declaration system to be used, as this choice imposes requirements on the seller and the carrier.

This article addresses the lack of flexible execution mechanisms in existing blockchain-based approaches for collaborative business process execution. In this perspective, the article proposes three types of controlled flexibility mechanisms: (i) dynamic binding of actors to roles of a collaborative process; (ii) dynamic selection of sub-processes; and (iii) dynamic selection of alternative pathways in a given execution state of a process. The first mechanism addresses the first of the above flexibility requirements while the latter two mechanisms address the second flexibility requirement. In order to enable participants to retain control, these flexibility mechanisms are associated with policies that determine which participants can initiate or have a say in a runtime decision, and what level of consensus needs to be achieved in such decisions. The article also proposes an approach to analyze policy specifications for dynamic role binding, in order to prevent circular dependencies that may prevent one or more roles to be bound to an actor under some circumstances. Finally, the article shows how the proposed policy specifications can be compiled into smart contracts that, once deployed on a blockchain platform, ensure that the actors exercise the flexibility captured in a collaborative process model within the boundaries set by the policies.

The proposed approach has been implemented in Caterpillar [6] – a blockchain-based execution engine for collaborative business processes[1]. The article reports on an experimental evaluation aimed at assessing the overhead of the proposed flexibility mechanisms and their associated policy enforcement in the context of the Ethereum blockchain.

This article is an extended and revised version of a conference paper [7]. The latter paper focused on the role-binding model and associated (role) binding policy specification language. With respect to the conference version, the main extensions in this article are:

- The control-flow flexibility mechanisms and their associated control-flow agreement model and policy specification language, as well as an approach to compile these policies into executable code.
- An extension of the policy definition language introduced in the conference paper, in order to enable roles to be bound based on a voting mechanism, instead of (or in addition to) requiring endorsement from a fixed set of actors.

The rest of the article is structured as follows. Section 2 introduces basic concepts of blockchain technology and discusses existing approaches for role binding and for handling flexibility in dynamic collaborative processes. Section 3 describes the role-binding model and its associated policy language. Subsequently, Section 4 presents the control-flow flexibility mechanisms and the associated agreement model and policy language. Section 5 discusses the semantics of the proposed policy languages and presents a verification approach to detect circular dependencies in role binding policies. Finally, Section 6 discusses the implementation and experimental evaluation, while Section 7 draws conclusions and sketches future work.

2. Background and Related Work

2.1. Blockchain Technology and Collaborative Processes

A blockchain is a distributed append-only store of transactions distributed across computational nodes and structured as a linked list of blocks, each containing a set of transactions [8]. A blockchain network is made up of nodes, a subset of which holds a replica of the data structure. Clients use

[1]https://github.com/orlenyslp/Caterpillar

a blockchain system (a concrete network) by reading data from and submitting transactions to it. Submitted transactions are grouped into blocks, which are broadcast across the network to be appended to the blockchain. To be accepted, a transaction must be properly formed and signed by their creator. No trust in individual clients or nodes is required, as the transactions are cryptographically signed, validated, and broadcast to the entire network. A consensus mechanism ensures tamper-proofness without assuming mutual trust between participants. In public blockchains like Ethereum, the full distribution of the transactions among untrusted nodes guarantees that it is almost impossible to tamper with the system. Characteristics like public verifiability, transparency, and integrity to prevent unauthorized modifications make these blockchains powerful in the presence of untrusted participants. However, these networks have performance problems, as the transaction throughput is limited, and the latency is high as a result of the mining process [9, 10].

A smart contract is a program deployed on the blockchain, which may be invoked via a transaction [8]. In Ethereum, smart contracts are typically written in the Solidity language, which is compiled into bytecode and executed on the Ethereum Virtual Machine. The computational and data storage consumption of a transaction is measured in *gas*, which translates to monetary costs for the transaction's sender. Each block has a *gas limit* and hence gas consumption directly impacts throughput. Contracts are deployed through transactions, with a gas cost that is largely proportional to the bytecode size. During deployment, a smart contract is assigned a unique address, used by client applications to call its functions. Such execution is also done by means of transactions, whose costs in gas depend on the complexity of the operations performed. External actors must hold an Ethereum account to deploy and interact with smart contracts. An account comprises a public address, i.e., a sort of user ID, which doubles as public key, and a private key to sign the transactions [11].

Existing blockchain-based process management tools support the specification of collaborative processes using BPMN [6, 12] or domain-specific languages [13], and their execution via smart contracts. These systems focus mainly on the control-flow perspective. Lorikeet [12, 14] and the tool proposed in [15] implement static access control mechanisms, where roles are bound to accounts upon process instantiation. A method proposed in [4] allows dynamic handoffs of process instances between actors, but does not support the specification and enforcement of permitted handoffs. Finally, the work presented in [16] proposes an interpreted execution of process models relying on dynamic data structures that allows updating the process at runtime. However, the approach has no restriction about who performs the updates.

2.2. Binding and Delegation Models for Collaborative Processes

A recent survey [17] found that only 30% of the Business Process Management Systems (BPMSs) analyzed support inter-organizational processes. The authors highlight, as one of the main challenges, the lack of trust between participants. At the same time, collaborative processes need to satisfy competing demands coming from the involved participants and hence need to incorporate some execution flexibility [18]. This is particularly the case in open and dynamic environments, where the actors may vary from one process instance to another and may even change during the execution of one process instance. Given these requirements, existing rule-based automated resource allocation mechanisms [19, 20] are not suitable, as these mechanisms assume that the rules for determining which resource will perform a given task can be determined upfront, at design-time. Accordingly, in this paper, we follow an alternative approach wherein the allocation of resources (actors in our context) to roles is determined entirely at runtime, based on consensus between actors. In other words, we adopt a dynamic role binding approach.

The idea of dynamic role binding has previously been considered in the context of Web service composition. For example, in the Business Process Execution Language (BPEL) [21], role binding is supported via *partner links*. A partner link is a variable that holds a reference to a service endpoint. This variable can be modified anytime during an execution of a process. This approach assumes that the whole process is orchestrated by a single actor and that this actor unilaterally decides which actor (i.e. endpoint) should be bound to each role (i.e. partner link). This assumption is also made in [22, 23]. A task-activity-based access control (TBAC) model, presented in [24], combines activities and dynamic permissions related to tasks in a business process. However, these approaches are not applicable in settings where the binding of actors to roles is not determined by a single actor.

Other studies have considered the problem of dynamic role binding in processes that are not orchestrated by a single actor. [25] extracts dynamic authorization policies from service choreographies. These policies are enforced locally by each party, but a central authority specifies all role bindings. BPEL4Chor [26] allows an actor to bind other actors to the roles it has control over. But each role is controlled by a single actor. In other words, collaborative role binding is not supported, e.g. this approach does not support the scenario where both the buyer and seller must agree on the actor who plays the role of carrier. Also, BPEL4Chor does not support role re-binding. In [27, 28], dynamic role bindings in decentralized processes are captured via delegations and revocations. This approach supports un-binding (revocation) but does not support collaborative binding (each actor decides on the roles it has control over).

In summary, none of the above studies has addressed the problem of dynamic role binding and un-binding in decentralized and dynamic processes, where multiple actors must collaboratively agree on each decision.

2.3. Flexibility and Dynamic Execution of Collaborative Processes

Flexibility is needed, for example, to support dynamic process adaptations in case of exceptions, and its evolution over the time [29]. Runtime adaptation is the ability to deviate execution of a specific process instance, while evolution refers to permanent changes to the process, i.e., affecting all the future instances. Other forms of flexibility point out variability as the possibility to maintain different variants of the same business process, and looseness when parts of the process model can be specified at runtime (also known as built-in flexibility) [29, 30].

Many different solutions support flexibility in the domain of process-aware information systems. Among those, [31] addresses the problem of runtime adaptation under the assumption that unexpected situations can be characterized by some known contextual elements. Others use planning strategies for automating business process reconfiguration at runtime [32], or automate the construction of exclusive choices considering multiple paths under a set of specific variable conditions [33]. Klingemann [34] introduces flexible elements into the workflow specification to fulfill goals restricted by a controlled set of runtime conditions. The works [35, 36] exploit the notion of worklets, i.e., self-contained sub-processes aligned to process tasks, to support the process evolution and adaptation at runtime. However, these solutions, like the others we found in the literature, mostly focus on the automation and validation of the flexibility, keeping aside the scenarios where trust is an issue, which makes them unsuitable for many inter-organizational processes.

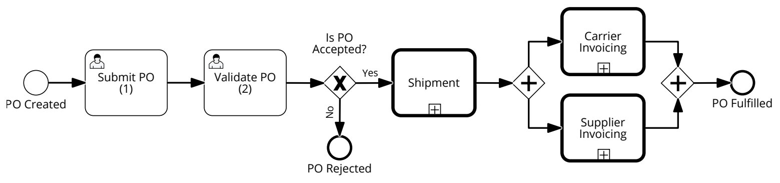
Partners in a collaborative setting may be suspicious about changes during the process execution. Indeed, a partner may gain an unfair advantage by arbitrarily changing the model [2]. Hence, existing blockchain-based approaches commonly use immutability to enforce conformance with a fixed implementation of the process, avoiding all kinds of flexibility during the process execution [3]. However, inter-organizational processes unavoidably involve tasks performed privately or requested by some party outside of the blockchain, and thus subject to trust issues. The latter introduces the need for using off-chain consensus among the participants of the collaboration, e.g., to approve a process update. Then the execution of the process can continue on-chain where the transactions are also enforced by consensus but among computers in the blockchain network.

Unlike existing approaches, this paper introduces the concept of *flexibility by consensus*, such that untrusting participants can validate the updates of the process at runtime. To that end, the role-binding and agreement policies described in this paper exploit the concepts of late-binding [37] and worklets [35] in order to prevent inconsistencies resulting from the process updates. Specifically, we follow an approach of flexibility by underspecification [37], under the basis of looseness and adaptation, where the full specification occurs at runtime and may vary with each instance [38].

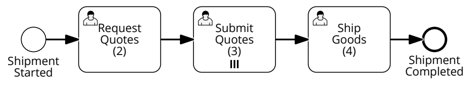
3. Dynamic Role Binding

The starting point of the proposed approach is a (collaborative) business process model where each task is associated with a role. For a given process instance (herein called a *case*), each role may be assigned to at most one actor. An actor has an identity (e.g., a blockchain account) and may represent a user, a group, an organization, a system or a device. As a running example, Fig. 1 shows a BPMN model of an order-to-cash process. There are six roles represented by numbers below each task label: (1) Customer, (2) Supplier, (3) Carrier-Candidate, (4) Carrier, (5) Invoicer and (6) Invoicee. Initially, a customer submits a purchase order (PO) to a supplier. If the PO is rejected the process terminates. Otherwise, the execution continues with the Shipment sub-process, where a supplier requests quotes from multiple carrier candidates (cf. the multi-instance task). Once the shipment completes, two parallel paths are taken to handle the payments. These payments are encapsulated in sub-process Invoicing. This sub-process is called twice: for the supplier's invoice and for the carrier's invoice.

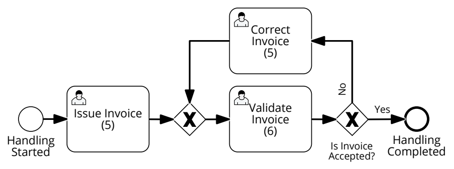
The act of assigning an actor to a role within a case is called *binding*. When a role is not assigned to an actor in a case, we say that the role is



(a) Root process: Order-to-Cash



(b) Sub-process: Shipment



(c) Sub-process: Invoicing

Figure 1: Running example: (1a) An *Order-to-cash* process linked, via call activities, to two reusable sub-processes; (1b) *Shipment* and (1c) *Invoicing*.

unbound. The binding of an actor to a role may happen anytime during a case. Actors may also be unbound from a role – an operation called *release*. A task is performed by the actor bound to the task's role. If a task is enabled when its associated role is unbound, the task waits until the role is bound. Actors may *nominate* themselves or other actors to play a role in a case, or they may request to release themselves or other actors from a role. Given the lack of trust, the nomination/release of an actor to/from a role may require the endorsement of actors playing other roles. If an actor is nominated to a role in a case, this nomination only leads to a binding if the required endorsements are granted. The *binding policy* of a process determines which role(s) are allowed to nominate an actor to a role, to request an actor's release from a role, and to endorse a nomination/release request.

3.1. Binding Policy Specification Language

A policy consists of a set of roles and a set of statements restricting how an actor may be nominated/released to/from a role. A statement is formed by a nominator, a nominee, and optionally a binding and/or an endorsement constraint. The nominator is a role that nominates/releases the actors of another role, namely the nominee. A binding constraint is a boolean expression stipulating that the nominee must be bound (or not) to an actor who is also bound to some other role(s). Binding constraints allow us to implement common resource allocation patterns such as segregation of duties and binding of duties [39]. An endorsement constraint is an expression that determines which roles need to endorse a nomination/release request. A role may be associated with the `case-creator`, implying that the role is bound upon case creation and does not need a nomination or endorsement. A policy statement applies by default to the root process, but it can be scoped to a sub-process call activity. Fig. 2 shows an extract of the grammar of the policy language in Backus Naur Form (BNF).[2]

```
⟨statement⟩ ::= [Under ⟨subprocess⟩ ',' ] ⟨role⟩ ⟨binding_expr⟩ [ ⟨endorse_constraint⟩ ] ';'
             |  ⟨role⟩ is 'case-creator' ';'

⟨binding_expr⟩ ::= ('nominates' | 'releases') ⟨role⟩ [⟨binding_constraint⟩]

⟨binding_constraint⟩ ::= ('in' | 'not in') ⟨set_expr⟩

⟨endorse_constraint⟩ ::= 'endorsed-by' ⟨set_expr⟩
                      |  'with' ⟨vote_ratio⟩ 'votes' ['by' ⟨role_list⟩]

⟨set_exp⟩ ::= ⟨role⟩
           |  ⟨role⟩ ('and' | 'or') ⟨set_expr⟩
           |  '(' ⟨set_exp⟩ ')'

⟨vote_ratio⟩ ::= ⟨floating_number⟩

⟨role_list⟩ ::= ⟨role⟩
             |  ⟨role⟩ ',' ⟨role_list⟩
```

Figure 2: BNF grammar describing the basic statement syntax of a binding policy.

Listing 1 shows a policy for the model in Fig. 1. The policy states that the case creator is automatically bound to the `Customer` role. The Customer nominates the `Supplier` (no endorsement needed here). The `Supplier`, in turn, nominates the `Candidate` (i.e., the carrier candidate) and the `Carrier`. The `Carrier` must be among the actors bound to the `Candidate` role (cf. binding constraint "Carrier in Candidate"). Note that `Candidate` is a role associated with a multi-instance task (`Submit Quotes`), implying that multiple actors may be bound to this role. The `Customer` must endorse the nomination of the `Carrier`. Under the Carrier Invoicing call activity, the `Invoicer` is nominated by the `Carrier` with an endorsement from the `Supplier` and `Customer`, and reciprocally for the `Invoicee`. Meanwhile, under the Supplier

[2]Some details (e.g. path expressions to refer to nested subprocesses) are omitted for space reasons and can be found at `http://git.io/caterpillar`.

```
{
 Customer is case-creator;
 Customer nominates Supplier;
 Under Shipment, Supplier nominates Candidate;
 Under Shipment, Supplier nominates Carrier in Candidate endorsed-by
     Customer;
 Under Carrier_Invoicing, Carrier nominates Invoicer endorsed-by Supplier
     and Customer;
 Under Carrier_Invoicing, Customer nominates Invoicee endorsed-by Carrier;
 Under Supplier_Invoicing, Supplier nominates Invoicer endorsed-by Customer;
 Under Supplier_Invoicing, Supplier nominates Invoicee endorsed-by Customer;
}
```

Listing 1: Binding Policy to control the execution of the processes modeled in Fig. 1.

Invoicing activity, the `Supplier` nominates the `Invoicer` with `Customer` endorsement, and reciprocally for the `Invoicee`.

This example illustrates the possibilities offered by the policy language to deal with lack of trust. For example, dishonest suppliers could try to derive benefits by not selecting the best carrier candidate but their preferred one. However, the customer would be able to reject such nominations. Also, the policy prevents the supplier from selecting a carrier that has not been a carrier candidate before.

The policy language also allows us to state that the set of actors who endorse a nomination request must fulfil a boolean expression. For instance, the above policy requires that both the buyer and the supplier must endorse the `Invoicer` of the carrier services. This scenario is relevant in the context of international trade, where both buyers and suppliers need to ensure that they do not deal with black-listed entities or entities in countries banned from trading. The boolean expressions in the endorsement constraint may contain arbitrary combinations of conjunctions and disjunctions. They may not, however, contain negation; e.g., it is not possible to state that the nomination is approved if a given actor refuses to endorse it. Such scenarios are not applicable in this setting.

Endorsement constraints can be specified as a ratio expression, which defines the percentage of votes needed for the statement to be accepted, and which roles can vote. The voting ratio (see grammar) is a float number between 0 and 1, i.e., from no votes needed to everyone must accept. The percentage is calculated based on the set of voters included in the statement. If no set of voters is specified, all participants assigned to a role are voters. Ratio expressions are less restrictive than boolean expressions as they rely

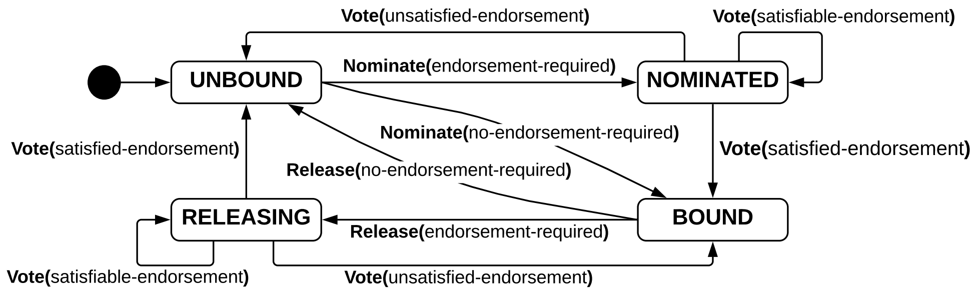

Vote(unsatisfied-endorsement)
Vote(satisfiable-endorsement)
UNBOUND
Nominate(endorsement-required)
NOMINATED
Nominate(no-endorsement-required)
Vote(satisfied-endorsement)
Release(no-endorsement-required)
Vote(satisfied-endorsement)
RELEASING
Release(endorsement-required)
BOUND
Vote(satisfiable-endorsement)
Vote(unsatisfied-endorsement)

Figure 3: Lifecycle of a role within a case.

on the amount instead of who is casting the votes. We refer the readers to Section 4 to see an example using ration expressions.

3.2. Runtime Role-Binding Operations

The role binding model relies on three operations. The `nominate` operation allows an actor to request that another actor (or itself) be bound to a role within a process instance (herein called a *case*). Inversely, a `release` operation allows an actor to request that another actor (or itself) be unbound from a role. The `vote` operation allows an actor to accept/reject a nomination or release request.

These operations trigger transitions in the *role lifecycle* depicted in Fig. 3. Within a case, a role is initially UNBOUND. After a `nominate` operation, the role changes to NOMINATED if it requires to be endorsed, otherwise is considered BOUND. A role in NOMINATED state, can transition to the BOUND state after a `vote` operation where the endorser accepts the nomination if, as a result of it, the endorsement constraint of this role is satisfied. On the contrary, a `vote` operation where the endorser rejects the nomination and by doing so makes the role's endorsement constraint unsatisfiable, triggers a transition to the UNBOUND state. If after a `vote` operation, the endorsement constraint remains satisfiable, then the role remains in the NOMINATED state. Symmetrically, a role can transit from BOUND to UNBOUND as a result of a `release` operation, via a RELEASING state, which is specular to the NOMINATED state. If the endorsement constraint associated to a *release* request becomes unsatisfiable, the role goes back to the BOUND state, and if it becomes satisfied, the role moves to the UNBOUND state.

Every binding of an actor to a role is made within a certain case scope, which is defined by a pair (*role*, *p-case*, where *p-case* is the identifier of an instance of the root process, a sub-process, or an activity.

At any given point in time, a role can be bound to at most one actor within a case scope. Binding an actor to a role within a child sub-process (i.e., a child case scope) hides nominations made for this role within any ancestor of the sub-process. For example, consider in Fig. 1 that *p-case*[O2C] is an instance of the root process ORDER TO CASH, and that the execution flow has reached the point in which one instance *p-case*[GS] of the sub-process GOODS SHIPMENT has already been created. Binding an actor A_1 to the case scope (`Supplier`, *p-case*[O2C]), implies that A_1 can perform the tasks `Validate PO` and `Request Quotes` in *p-case*[O2C] and *p-case*[GS] respectively. However, binding a new actor A_2 in the case scope (`Supplier`, *p-case*[GS]) allows A_2 to perform `Request Quotes`, and restricting A_1 to perform only `Validate PO`. Importantly, case scopes are defined with respect to identifiers of process, sub-process, or task instances. In the context of a multi-instance sub-process or a multi-instance task, each instance of this sub-process or task defines a new case scope for each role. Within each of these instances, an actor may be bound to a role, and the actor bound to a given role R may differ from one instance to another.

Note that the ability to bind an actor to a role within a given case scope may be restricted by a binding policy. In this respect, the keyword `Under` in the binding policy specification language allows one to restrict how an actor may be bound to a role within a given sub-process of the process hierarchy. Case scopes apply to the execution of tasks. In the case of binding and endorsement constraints, all the actors bound to a role across the whole process hierarchy are eligible, no matter in which case scope they were bound.

Binding an actor into a role follows the rules in Definition 1. Subsequently, Definition 2 describes how to assert if an actor can perform a task.

Definition 1 (Runtime Rules). *Consider the actors* `nominator`*,* `nominee` *and* `endorser`*, who can respectively play the roles* `r-nominator`*,* `r-nominee` *and* `r-endorser` *in a process instance* `p-case`*.*

1. `nominator` *can nominate* `nominee` *in* `p-case` *iff* [3]*:*

(a) *An actor is* BOUND *as* `case-creator` *in the hierarchy containing* `p-case`*,*

[3] if an only if

   (b) *the state of* `r-nominee` *in* `p-case` *is* UNBOUND,

   (c) *the policy asserts that* `r-nominator nominates r-nominee`*, and* `nominator` *is* BOUND *as* `r-nominator` *in some case scope in the hierarchy containing* `p-case`.[4] *Besides, if a binding constraint is defined in the statement, it must be fulfilled based on the roles held by* `nominee` *at the moment of nominating.*

   (d) *The nomination of the* `case-creator` *is independent of the previous rules, but* `p-case` *must be root in the process hierarchy. Besides, no actor could be bound to any case scope in hierarchy containing* `p-case` *before. The nomination of a case creator requires no endorsement and cannot be released. Accordingly, the state is updated to* BOUND *after the operation.*

2. `nominator` *can release* `nominee` *in* `p-case` *iff:*

   (a) `nominee` *is* BOUND *as* `r-nominee` *in* `p-case`.

   (b) `nominator` *is* BOUND *as* `r-nominator` *in some case scope in the hierarchy containing* `p-case`.

   (c) *The policy asserts that* `r-nominator releases r-nominee`. *Besides, if a binding constraint is defined in the statement, it must be fulfilled given the roles held by* `nominee` *at the moment of releasing.*

3. `endorser` *can vote for a nomination/release of* `nominee` *in* `p-case` *iff:*

   (a) `nominee` *is* NOMINATED *as* `r-nominee` *in* `p-case` *for voting about a nomination, or* RELEASING *for voting about a release operation.*

   (b) `endorser` *is* BOUND *as* `r-endorser` *in some case scope in the hierarchy containing* `p-case`.

   (c) `endorser` *can vote to accept or reject once. Besides,* `r-endorser` *is included in an endorsement constraint of the statement of the operation.*

[4]Case scopes are defined to restrict the execution of tasks. Thus, checking that `nominator` is BOUND as `r-nominator` should consider not only the ancestors of `p-case` but the full hierarchy. The same logic applies to the endorsers.

Definition 2 (Access Control to Perform Tasks). *A task* `T` *enabled in a process instance* `p-case` *can be performed by an actor* `A` *iff:*

1. *One role, namely* `R`*, is related to* `T` *in the process model.*
2. *Then, say* `S` *is the closest case scope from* `p-case` *to any ancestor in the process hierarchy, such that* `R` *is* Bound*. It must hold that* `S` *exists, and* `A` *is the actor appointed in* `S`*.*

4. Control-Flow Flexibility and Agreement Policies

Role-binding policies offer a dynamic schema on the resource perspective but do not address how to manage updates on the control-flow perspective at runtime. Accordingly, the actors who are bound to a role must collectively decide how to proceed, which leads to an extension of the role-binding schema with agreement policies.

We propose agreement policies that define, in a certain scope, which actors can participate, and reach consensus to update the control-flow perspective at runtime. For example, consider that, during the execution of a case of the process modelled in Fig. 1, one `candidate` made a mistake when submitting the quotes. Accordingly, the `supplier` would like to allow him to fix it before making the final decision. However, no task exists to that end in the control-flow, and allowing to roll back the process state could introduce inconsistencies. Late-binding of a non-interrupting event sub-process (see Fig. 4), running in parallel with the current process case [40], allows the supplier to decide, for the current instance, which tasks are required to fix the issue before proceeding with the execution. A first approach to allow the late-binding can require that all the participants agree on it. However, such action in the example mainly involves the `supplier` and the `candidates`, so it will introduce some extra and unnecessary responsibilities to the other participants. Instead, an agreement policy can include a statement granting that in the subprocess Shipment, a `candidate` can link a sub-process if the `supplier` agrees.

4.1. Agreement Policies on Control Flow

An agreement consists of a set of statements restricting how an actor, bound to a role in a given process instance, can act to update control-flow elements, e.g., sub-processes, tasks, gateways, on the corresponding process

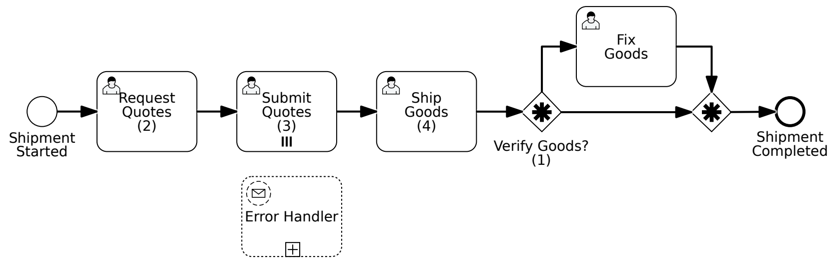


Figure 4: A more flexible variant of the sub-process Goods Shipment displayed in Fig. 1.

model at runtime. A statement is formed by a requester, an action constraint, and optionally an endorsement constraint. The requester is a role that requests an action to perform on a control-flow element at runtime represented by an action constraint. Endorsement constraints work like in the role-binding policies and determine which roles can endorse the request. Besides, a policy statement applies by default to the root process, but it can be scoped to sub-processes or call activities. Fig. 5 illustrates an extract of the grammar of the policy language in Backus Naur Form (BNF).[5]

⟨*statement*⟩ ::= [Under ⟨*subprocess*⟩ ','] ⟨*role*⟩ 'can' ⟨*action_constraint*⟩ ⟨*endorsement_constraint*⟩ ;

⟨*action_constraint*⟩ ::= ⟨*action*⟩ '`on`' ⟨*control-flow_element*⟩

⟨*action*⟩ ::= ('`link-process`' | '`link-role`' | '`choose-path`')

Figure 5: BNF grammar describing the basic statement syntax of an agreement policy.

The agreement policies provide controlled flexibility relying on three actions. The first two actions supported are the late-binding of sub-processes and roles via the actions `link-process` performed on call-activities and collapsed sub-processes, and `link-role` targeting user and service tasks. Besides, we use dynamic gateways (i.e., complex gateways in BPMN) to allow actors deciding on which outgoing flow arcs to move during the process execution via the action `choose-path`. Accordingly, the activation conditions in the dynamic gateways are driven by agreement policies that rely on user decisions instead of internal conditions that verify the process data.

[5]Some details (e.g., path expressions to refer to nested subprocesses) are omitted for space reasons and can be found at `http://git.io/caterpillar`.

The rationale for selecting the three flexibility mechanisms proposed in this paper is the following:

- `link-process` exploits concepts extensively addressed in the literature on flexibility in workflow systems, such as worklets [35, 41], pockets of flexibility [38, 42], late binding and late modelling [43, 44, 45]. In these approaches, participants can define or reuse parts of the process at runtime. In the blockchain setting, every sub-process, or process fragment, is mapped into a smart contract derived from a process model. Then, the agreement policies offer the set of rules for the participants to decide by consensus which smart contract to bind at runtime.

- `link-role` naturally enhances role-binding policies. This operation removes the need for tagging every task of the process model with a role at design time. Instead, the process participants can dynamically decide by consensus not only which actors can play a role but also the association of roles to tasks at runtime.

- `choose-path` complements the decision rules on the gateways. Traditional approaches use decision rules based on case data to choose among the outgoing flow arcs. However, in collaborative processes, the data required to make a collective decision is not always accessible, as parties do not wish to disclose the data to each other, and they often have conflicting interests. Thus, oftentimes, the decisions on how to proceed must collectively be agreed by the participants, rather than being taken based on data available to all parties.

The proposed approach could be extended with other flexibility mechanisms, such as adding, skipping or removing elements in the process model. We note, however, that these latter flexibility mechanisms may lead to deadlocks. Checking for the possibility of deadlocks on-chain may be prohibitively expensive in terms of computation. In this paper, we focus on the above three flexibility mechanisms, which do not introduce deadlocks and therefore do not require additional verification techniques to be put into place.

Listing 2 illustrates an agreement policy extending the role-binding policy in Listing 1, and related to the models in Figs. 1 and 4. The first statement describes how the `supplier` can perform a late-binding of the call activity SHIPMENT in the root process ORDER TO CASH if all the bound roles agree on it. Here, `supplier` could decide, for example, between the sub-process in

```
{
  Supplier can link-process on Goods_shipment with  1.0 votes;
  Under Shipment, Candidate can link-process on Error_Handler endorsed by
      Supplier;
  Under Shipment, Customer can choose-path on Verify_Goods with 0.5 votes by
       Supplier, Carrier;
  Under Shipment, Customer can link-role on Fix_Goods with 0.5 votes by
      Supplier, Carrier;
}
```

Listing 2: Agreement Policy to support the execution of the processes modeled in Figs. 1 and 4.

Fig. 1b, or the one in Fig. 4 which offers a more flexible execution. Besides, if we assume that only the `customer` should be bound at the moment of linking the sub-process, only his/her vote is required. The second statement is scoped to the subprocess SHIPMENT to allow a `candidate` to link the event sub-process ERROR HANDLER if the `supplier` agrees. This statement is aligned with the example we presented above if a `candidate` makes a mistake when submitting the quotes. From the BPMN standard, non-interrupting event sub-processes are enabled and can run in parallel to the (sub-)process where they are enclosed. Thus, late-binding of event sub-processes can be exploited to handle exceptions at runtime, e.g., for the `candidate` to fix the wrong quotes. The last two statements, scoped to the SHIPMENT sub-process in Fig. 4, allows the verification of the goods and solving possible issues after the delivery in the off-chain world. Specifically, the dynamic gateway allows the `customer` to decide how to proceed after the delivery based on the quality of the goods received. Note that the responsibility for an eventual problem may be at the `supplier` or at the `carrier`. Thus, one of them must accept/respond to the decision of the `customer`. Besides, the late-binding of a role to the task FIX GOODS allows appointing at runtime the party responsible for the problem to solve it.

4.2. Runtime Agreement Operations

The agreement policies rely on three operations. The `request` operation allows an actor to ask for an action to enforce a process case. A request includes the `action`, the target smart contract instance, i.e., the process case, and the metadata of the element to update. For example, to link a sub-process in a process case, as described by [6], the `request` must include the action `link-process`, the blockchain address of the process case, and the information required to update the element, e.g., a factory contract to

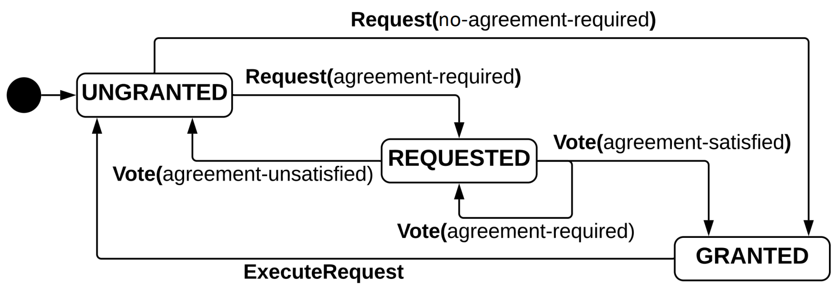

Request(no-agreement-required)
UNGRANTED
Request(agreement-required)
REQUESTED
Vote(agreement-satisfied)
Vote(agreement-unsatisfied)
Vote(agreement-required)
GRANTED
ExecuteRequest

Figure 6: Lifecycle of an action to be performed at runtime.

instantiate the sub-process and the hash identifying the compilation artifacts of the subprocess to link. The `vote` operation allows an actor to accept/reject a request. Finally, the `execute` operation enforces an (accepted) action as described in the request. Note that these runtime operations affect only the process case where they are triggered.

These change operations trigger transitions in the *action lifecycle* depicted in Fig. 6. Within a case, an action is initially UNGRANTED. After a `request` operation, the state changes to REQUESTED if it requires to be accepted by agreement, otherwise is considered GRANTED. An action in REQUESTED state can transit to the GRANTED state after a positive `vote` if, as a result of the vote, the agreement policy is satisfied. On the contrary, a negative `vote` might make the agreement policy unsatisfiable, and if so triggers a transition to the UNGRANTED state. Finally, if after a `vote` operation the agreement policy remains satisfiable, then the action remains in the REQUESTED state. An action in GRANTED state can be performed only once in the lifecycle. Thus, the `execute` operation moves the action state from GRANTED to UNGRANTED, i.e., re-executing the action in the future starts a new iteration of the lifecycle.

5. Policy Consistency Verification

Binding policies have the potential to be inconsistent with a process model, and the majority of this section is focused on such interdependencies. At the end of the section, we discuss the consistency of role-binding policies, agreement policies, and process models.

Nomination and release statements in a role-binding policy implicitly induce precedence dependencies in the binding of roles. A statement `R1 nominates R2 endorsed-by R3` implies that for R2 to be bound, R1 and

R3 must be bound before. Circular and unresolvable dependencies induced in this way may lead to deadlocks. Accordingly, we define a notion of policy consistency as follows. A policy is *consistent* if, starting from the state where only the roles associated with case-creator are BOUND and after executing any allowed sequence of nomination, release and endorse operations, we always reach a state where all roles will reach the BOUND state via some (other) sequence of nomination, release and endorse operations.

To verify policy consistency, we define a mapping from a policy to a Petri net [46], herein called a *nomination net*. Given the nomination net of a policy, we map the problem of checking policy consistency to a problem of reachability analysis over Petri nets. Algorithm 1 maps a policy to a nomination net. For the sake of conciseness, this algorithm focuses on nomination statements, leaving aside release statements. The mapping of release statements follows a similar structure. For the same reason, the algorithm leaves aside binding constraints.

To illustrate the algorithm, we consider the binding policy in Fig. 7. The algorithm takes as input a symbolic representation of a policy consisting of a set of roles and a set of tuples of the form (nominator, nominee, endorsement-constraint), with $\perp$ denoting an empty constraint. For example, the symbolic representation of the policy in Fig. 7 is given in Fig. 8. Given this input, the algorithm will produce as output the nomination net in Fig. 9.

The algorithm proceeds as follows. After initializing variable `RNets` in line 2, the algorithm builds a Petri net for each node in lines 3-4 (Step 1). Let us consider that we are building the Petri net for role A, which is shown in color blue in Fig. 9. In line 4, the algorithm creates such a Petri net with three places, namely u_A, n_A and b_A, which represent the states of the role's lifecycle UNBOUND, NOMINATED and BOUND, respectively. Similarly, two transitions are added to the Petri net, namely nm_A and en_A, representing the operations 'nominate' and 'endorse'. Finally, four arcs added to complete the Petri net, by connecting the places and transitions. The Petri nets for all the other nodes are created in a similar way. Every Petri net thus created is added to `RNets` that serves as a map that associates a role to its corresponding Petri net.

In lines 5-9 (Step 2), all the role (Petri) nets are merged to form the initial nomination net, which is held in variable `NNet`. This is done by taking the union of the elements in the role nets. Also, the initial marking is set to the empty set.

In lines 11-14 (Step 3), the algorithm adds double-headed arcs to the

Algorithm 1 Construction of the Nomination Net for a given Binding Policy

```
 1: function ConstructNominationNet(R, BP)
 2:     RNets ← ∅
    ▷ Step 1: Build a Petri net for each role
 3:     for each role $r \in \mathrm{R}$ do
 4:         $\mathrm{RNets} \leftarrow \mathrm{RNets} \bigcup \left\{ \left( r \mapsto \left\langle \begin{array}{ll} \{u_r, n_r, b_r\} & \triangleright P_r \\ \{nm_r, en_r\} & \triangleright T_r \\ \{(u_r, nm_r), (nm_r, n_r), (n_r, en_r), (en_r, b_r)\} & \triangleright F_r \end{array} \right\rangle \right) \right\}$
    ▷ Step 2: Merge all role nets to form the nomination net
 5:     let $\mathrm{NNet} = \langle P, T, F, M_0 \rangle$ in
 6:         $P \leftarrow \bigcup_{r \in \mathrm{R}} \mathcal{P}(\mathrm{RNets}[r])$
 7:         $T \leftarrow \bigcup_{r \in \mathrm{R}} \mathcal{T}(\mathrm{RNets}[r])$
 8:         $F \leftarrow \bigcup_{r \in \mathrm{R}} \mathcal{F}(\mathrm{RNets}[r])$
 9:         $M_0 \leftarrow \emptyset$
10:
    ▷ Step 3: Wire up operation nominate
11:     for each $\langle r_{nr}, r_{ne}, \_ \rangle \in \mathrm{BP}$ do
12:         select $b_{nr} \in \mathcal{P}(\mathrm{RNets}[nr])$
13:         select $nm_{r_{ne}} \in \mathcal{T}(\mathrm{RNets}[ne])$
14:         $\mathcal{F}(\mathrm{NNet}) \leftarrow \mathcal{F}(\mathrm{NNet}) \cup \{(b_{r_{nr}}, nm_{r_{ne}}), (nm_{r_{ne}}, b_{r_{nr}})\}$
    ▷ Step 4: Wire up operation endorse
15:     for each $\langle r_{nr}, r_{ne}, eex \rangle \in \mathrm{BP}$ such that $eex \neq \bot$ do
16:         $\mathcal{P}(\mathrm{NNet}) \leftarrow \mathcal{P}(\mathrm{NNet}) \cup \{disj_{r_{ne}}, eex_{r_{ne}}\}$
17:         $\mathcal{F}(\mathrm{NNet}) \leftarrow \mathcal{F}(\mathrm{NNet}) \cup \{(nm_{r_{ne}}, disj_{r_{ne}}), (eex_{r_{ne}}, en_{r_{ne}})\}$
18:         for each conj $\in eex$ do
19:             $\mathcal{T}(\mathrm{NNet}) \leftarrow \mathcal{T}(\mathrm{NNet}) \cup \{eex_{conj}\}$
20:             $\mathcal{F}(\mathrm{NNet}) \leftarrow \mathcal{F}(\mathrm{NNet}) \bigcup_{r \in conj \wedge b_r \in \mathcal{P}(\mathrm{RNets}[r])} \left\{ \begin{array}{l} (b_r, eex_{conj}), (eex_{conj}, b_r), \\ (disj_{r_{ne}}, eex_{conj}) \end{array} \right\}$
    ▷ Step 5: Update NNet's initial marking
21:     let $r_{cc} \in R$: $r_{cc}$ be case creator in
22:         $\mathrm{Ps} \leftarrow \{u_r \mid r \in R \setminus \{r_{cc}\} \wedge u_r \in \mathcal{P}(\mathrm{NNet}[r])\} \cup \{b_{r_{cc}} \mid b_{r_{cc}} \in \mathcal{P}(\mathrm{NNet}[r_{cc}])\}$
23:         $M_0(\mathrm{NNet})(p) = \left\{ \begin{array}{ll} 1 & \text{if } p \in Ps \\ 0 & \text{Otherwise} \end{array} \right.$
24:
25:     return NNet
```

Petri net to synchronize the transition that represents the nomination of roles. To illustrate the idea of nomination, consider the double-headed arc connecting the place b_A and the transition nm_B in Fig. 9, highlighted in red. Simply put, role A will be able to nominate role B when role B is UNBOUND and role A is BOUND (b_A must hold a token). The firing of transition nm_B, that is "nominate B", will change the state of role B from UNBOUND to NOMINATED. The double-headed arc will keep a token in b_A after the nomination of role B.

The encoding of endorsement conditions is handled in lines 15-20 (Step 4). Without loss of generality, we assume that the endorsement conditions are expressed in disjunctive normal form, meaning that there is only one disjunction that relates several conjunctions. Besides, ratio expressions are

```
{ A is case-creator;
  A nominates B;
  A nominates C;
  C nominates D, endorsed-by A and B;
}
```

Figure 7: Sample binding policy

$$R = \{A, B, C, D\}$$
$$BP = \{\langle A, B, \perp\rangle, \langle A, C, \perp\rangle, \langle C, D, A \wedge B\rangle\}$$

Figure 8: Symbolic representation of the binding policy in Fig. 7

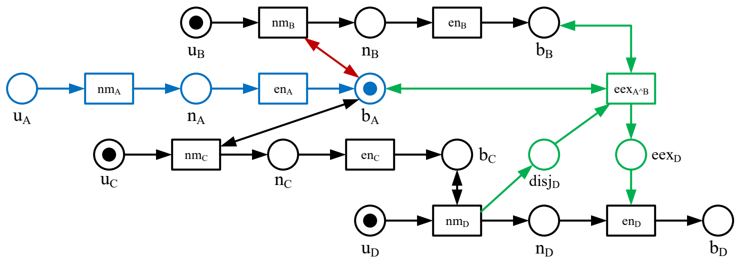


Figure 9: Nomination net for binding policy in Fig. 7

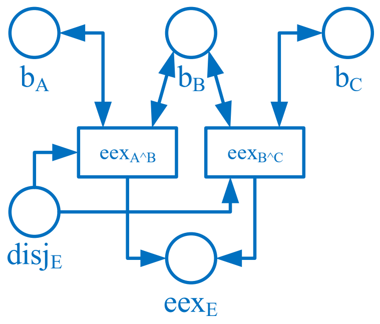


Figure 10: Net encoding condition $(A \wedge B) \vee (B \wedge C)$

considered as a single conjunction set. We consider two additional cases: (1) no endorsement condition is specified (represented by $\perp$), meaning that no endorsement is required, and (2) only one conjunction is specified. To illustrate this step of the construction of the nomination net, consider the binding policy:

D nominates E, endorsed-by (A and B) or (B and C);

The Petri net in Fig. 10 encodes the endorsement condition in the above policy: $(A \wedge B) \vee (B \wedge C)$. The latter is bound to variable eex in line 15.

In line 16, the algorithm adds two new places: $disj_E$ which encodes the disjunction, and eex_E, which collects the outcome of the endorsement (i.e.

```
{
 J is case-creator;
 J nominates K, endorsed-by L;
 J nominates L, endorsed-by K;
}
```

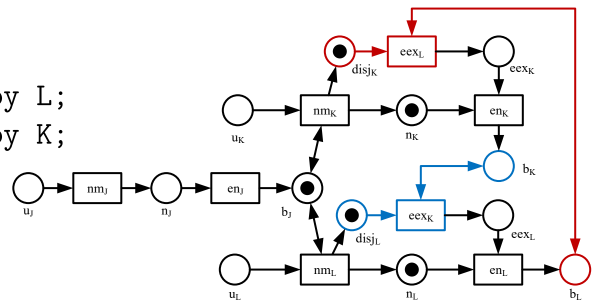


Figure 11: Binding policy with circular dependency and its nomination net

it holds a token when one of the endorsement conditions is met). In line 17, these are connected to the transitions of the role: from the nomination nm_E to $disj_E$, and from the outcome eex_E to the endorsement en_E (not shown in Fig. 10). Then, in line 18, the algorithm iterates over each one of the conjunctions. In line 19, a new transition, representing the underlying conjunction is added to the net, and the corresponding arc in line 20. For instance, the net in Fig. 10 has transition $eex_{A\wedge B}$ representing conjunction $A \wedge B$, and $eex_{B\wedge C}$ representing $B \wedge C$. Only $eex_{A\wedge B}$ or $eex_{B\wedge C}$ will be able to consume the token held by $disj_E$, which prevents the generation of an arbitrary number of tokens in `NNet`. $disj_E$ receives a token when nm_E fires, i.e., when D nominates E. The disjunction expressed in this way means that role E can be endorsed if at least one of the conjunctions holds true, which corresponds to the firing of one of the transitions $eex_{A\wedge B}$ and $eex_{B\wedge C}$. Returning to the example in Figures 7-9, we observe that role D is endorsed if and only if both roles A and B are BOUND. The subnet implementing the endorsement condition is shown in green in Fig. 9.

Finally, lines 21-23 set the initial marking for the nomination net. Briefly, line 21 will add a token to the place representing the state UNBOUND of every single role, except for the "case creator". In the latter case, we add a token to the place representing the state BOUND.

To verify policy consistency, we use reachability analysis to check if the marking where all roles are bound is always reachable starting from the initial marking where only the roles associated to `case-creator` are bound. In other words, there is no deadlock preventing a role from being bound. Fig. 11 shows a binding policy with a circular dependency, leading to a deadlock in the corresponding nomination net. Fig. 11 shows the marking where the deadlock occurs. Both roles K and L have been nominated by role

J. Hence, $disj_K$ has a token, but transition eex_L cannot fire until b_L has also a token. In order for b_L to have a token, however, transition eex_K needs to fire because it requires b_K to have a token.

In the discussion above, we focused on the verification of the consistency of role binding policies, which as we saw above, may contain circular dependencies that lead to deadlocks. Below, we argue that the proposed control-flow flexibility mechanisms and agreement policies are designed in such a way that they do not lead to deadlocks.

Each statement in an agreement policy is composed of a requester, an action constraint, and endorsement constraint. An agreement policy is consistent if each statement fulfils the following criteria:

- `Requester:` If the role of the requester is defined within a consistent role-binding policy, then it will always be possible to reach a state where an actor is bound to this role. Once this happens, this actor may act as the requester.

- `Action constraint:` An action constraint on `link-process` relates a call-activity to an instance of a smart contract implementing a sub-process. Assuming that the BPMN process model is semantically correct, there will be at least one execution path leading to the enablement of the call-activity. If every role associated to a task in the sub-process is defined within a consistent role-binding policy, these roles will eventually be bound to actors, and the sub-process will have bindings for all roles required to be executed. Similarly, a dynamic gateway is consistent if the roles involved in the evaluation of its associated conditions are part of a consistent role binding policy.

- `Endorsement constraint:` An endorsement constraint is consistent if every role it involves is defined within a consistent role binding policy. If this is the case, the corresponding roles will eventually be bound to actors and these actors will be able to provide their endorsement.

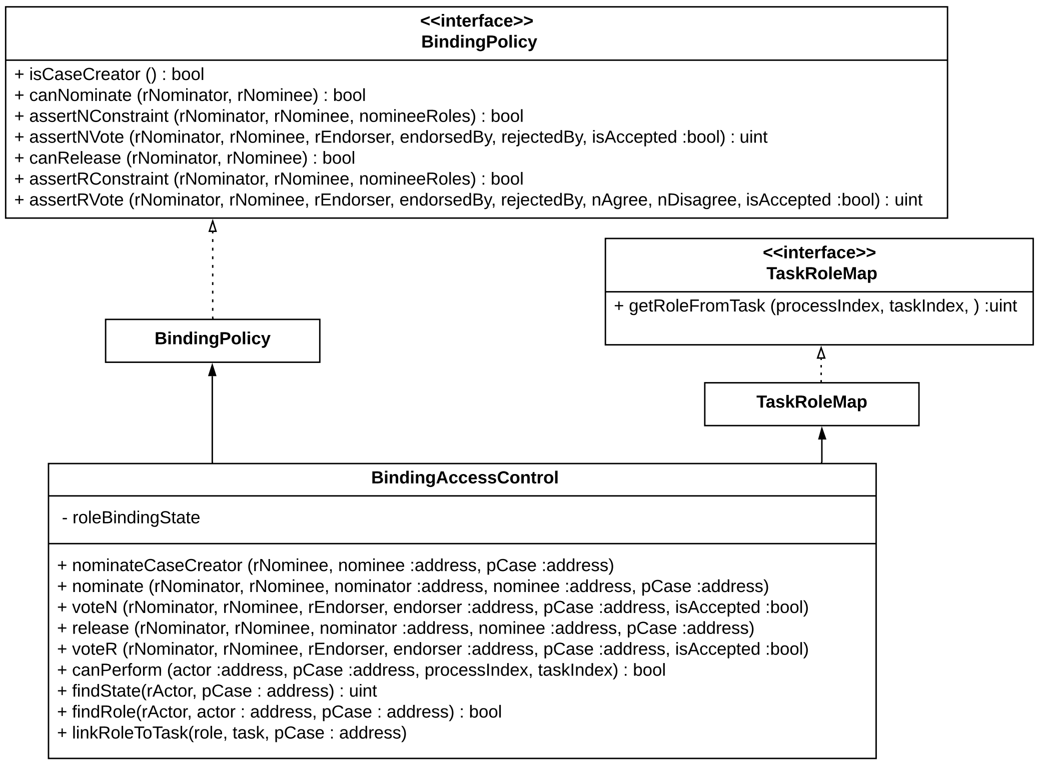
Summarizing, so long as the roles that need to participate in a control-flow decision have been bound to corresponding actors, it is always possible for these actors to reach agreement on which sub-process to execute or which branch of a dynamic gateway to choose. Hence, if the role-binding policy is consistent, and the control-flow agreement policy only refers to roles defined in the role binding policy, then the control-flow agreement policy is consistent as well.

6. Implementation and Evaluation

To demonstrate the proposal's feasibility, we developed a compiler that takes as input a policy specification (i.e., role-binding or agreement) and produces Solidity smart contracts to enforce the policy. This policy compiler is designed to be used in conjunction with the Caterpillar BPMN-to-Solidity compiler [6]. The smart contracts generated by the policy compiler manage the association between roles, actors (represented as blockchain accounts) and the requests of late-binding and dynamic gateways at runtime, while the smart contracts generated by the BPMN-to-Solidity compiler enforce the control-flow constraints in the process model. When a task is enabled, the *worklist handler* smart contract of Caterpillar checks if the corresponding role is bound to an actor within the current case, and ensures that only this actor can execute the task. The prototype allows, via REST interactions, the validation of binding policies that are compiled later into smart contracts. Besides, it supports to perform the runtime operations, i.e., nomination, release, request and vote, as well as executing process models restricted by our access control approach, and with the added flexibility of the agreement policies. The source code of Caterpillar, including the binding policy compiler and the examples used in this paper, are available at `http://git.io/caterpillar`. Below we discuss the generation of smart contracts and evaluate the costs generated by these contracts.

6.1. Compiling Role-Binding Policies into Smart Contracts

From a process model and a policy specification, the compiler generates a smart contract (named BindingPolicy) to encode the role-binding policy and one smart contract (TaskRoleMap) encoding the task-role relations in the process model. The BindingPolicy contract encodes the logic of who can nominate and release each role as well as the binding and endorsement constraints. A third contract (BindingAccessControl) implements the runtime operations sketched in Section 3. BindingPolicy, TaskRoleMap and BindingAccessControl are singleton contracts – only one instance of each of them is created since these contracts only maintain schema-level data. The BindingAccessControl contract maintains the state of each role in each process case, as per the life-cycle in Fig. 3. When a nomination, release, or vote operation is invoked, the BindingAccessControl contract invokes the BindingPolicy contract. The latter checks if this operation is allowed in the current state and computes the new state.



<<interface>>
BindingPolicy
+ isCaseCreator () : bool
+ canNominate (rNominator, rNominee) : bool
+ assertNConstraint (rNominator, rNominee, nomineeRoles) : bool
+ assertNVote (rNominator, rNominee, rEndorser, endorsedBy, rejectedBy, isAccepted :bool) : uint
+ canRelease (rNominator, rNominee) : bool
+ assertRConstraint (rNominator, rNominee, nomineeRoles) : bool
+ assertRVote (rNominator, rNominee, rEndorser, endorsedBy, rejectedBy, nAgree, nDisagree, isAccepted :bool) : uint
<<interface>>
TaskRoleMap
+ getRoleFromTask (processIndex, taskIndex,) :uint
BindingPolicy
TaskRoleMap
BindingAccessControl
- roleBindingState
+ nominateCaseCreator (rNominee, nominee :address, pCase :address)
+ nominate (rNominator, rNominee, nominator :address, nominee :address, pCase :address)
+ voteN (rNominator, rNominee, rEndorser, endorser :address, pCase :address, isAccepted :bool)
+ release (rNominator, rNominee, nominator :address, nominee :address, pCase :address)
+ voteR (rNominator, rNominee, rEndorser, endorser :address, pCase :address, isAccepted :bool)
+ canPerform (actor :address, pCase :address, processIndex, taskIndex) : bool
+ findState(rActor, pCase : address) : uint
+ findRole(rActor, actor : address, pCase : address) : bool
+ linkRoleToTask(role, task, pCase : address)

Figure 12: Class diagram of the smart contracts derived from the role-binding policies.

The class diagram in Fig. 12 captures the functionality of the generated smart contracts. Input parameters with no type specification are by default `uint`. As stated above, contract BindingAccessControl implements the runtime operations for nomination, release and voting. Since this contract does not encode anything about a particular policy, it is not generated by the policy compiler. However, instead, it is hard-coded and deployed once on the target Ethereum blockchain. This contract maintains the state of the role bindings for a given case in a variable called roleBindingState. Given that the cost of a smart contract depends on the amount of data it maintains, we encode the roleBindingState using bitmaps. Similarly, the endorsement constraints are represented as bit arrays. Specifically, we first put these constraints in disjunctive normal form, e.g., `(A and B and ...) or (D and ...)`. Then we implement each conjunction set as a bit array, and encode it as a 256-bits unsigned integer – the default word size in

Ethereum.[6] Besides, the contract BindingAccessControl provides the functions `findState`, `findRole` and `linkTaskToRole` to query the state of a role, to check if a given actor is bound to a role and to link a role to a task (via an agreement policy), respectively.

Contract TaskRoleMap is generated from the process model. This contract is straightforward (it maps tasks to roles), so we do not discuss it further. The role-binding policy specification is compiled into the BindingPolicy contract. These contracts, BindingPolicy and TaskRoleMap, were compiled statically, i.e., they do not store any dynamic information on the blockchain storage, what makes the policies immutable (once deployed), and avoids high costs derived from accessing the storage during the process execution. Below we discuss how the role-binding functions are generated (functions `canNominate`, `assertNConstraint` and `assertNVote`). The generation of the release functions (`canRelease`, `assertRConstraint` and `asserRVote`) is done in a similar way.

To generate function `canNominate`, for each distinct nominator in the policy a conditional and bit array, namely `nMask`, is created with one bit per role such that the presence of a nominee is represented with a *one* and the absence with a *zero*. For example, a nominator with index 3 and `nMask = 6` is translated into:

```
function canNominate(uint rNominator, uint rNominee) returns(bool) {
    ...
    if (rNominator == 3)
      return 6 & (1 << rNominee) != 0;
    ...
}
```

Function `assertNConstraint` verifies if the roles held by a nominee do not contradict the binding constraint. Thus, a conditional instruction is added per nomination statement that includes a binding constraint. A statement is identified by the union of nominator and nominee, i.e., `(1 << rNominator) | (1 << rNominee)`. Variable `nomineeRoles` is the bit array encoding the nominee's current roles. A constraint of the form `(A and B) or (C) or ...` is satisfied if at least one conjunction set is fully included in `nomineeRoles`. The latter is encoded as follows:

[6]Note that implementing the bitsets as 256-bits integer is not a limitation, because if the number of roles/elements is greater than 256, we can use a list of integers instead.

```
if ((1 << rNominator) | (1 << rNominee))
  return nomineeRoles & ((1 << A) | (1 << B)) == ((1 << A) | (1 << B))
       || nomineeRoles & (1 << C) == (1 << C) || ...;
```

Function `assertNVote` checks if an endorser can vote for a nomination and determines the state after this vote. In endorsement constraints written as boolean expressions, given the input parameters `endorsedBy` and `rejectedBy`, which are bit arrays encoding the roles that already accepted and rejected the nomination, this function determines the resulting state as follows:

1. Bound if all the roles in at least a conjunction set, namely `CS`, endorsed the nomination, i.e., `(endorsedBy | endorserRole) & CS == CS`,

2. Unbound if in each conjunction set contains at least one role rejected the nomination, i.e., for each `CS`, `(rejectedBy | endorserRole) & CS != 0`,

3. Nominated if none of the conditions 1. and 2. are fulfilled yet, i.e., there is at least a conjunction set with no rejections and with roles pending to vote.

In endorsement constraints written as a ratio expression, given the input parameters `nAgree` and `nDisagree`, which counts the number of roles that accepted or rejected the request, the function determines the resulting states as follows:

```
if(isAccepted && nAgree + 1 >= vRequired)
    return BOUND;
else if(!isAccepted && rTotal + rAgreed - nDisagree - 1 < vRequired)
    return UNBOUND;
return NOMINATED;
```

Where `rTotal` and `vRequired` are compiled from the agreement policy, and represent the total of roles allowed to vote and the number of votes required to accept the request.

6.2. Compiling Agreement Policies into Smart Contracts

The class diagram in Fig. 13 captures the functionality of the smart contracts generated from agreement policies. Input parameters with no type specification are by default `uint`. Contract DynamicProcessManager implements the runtime operations for request and voting as described in

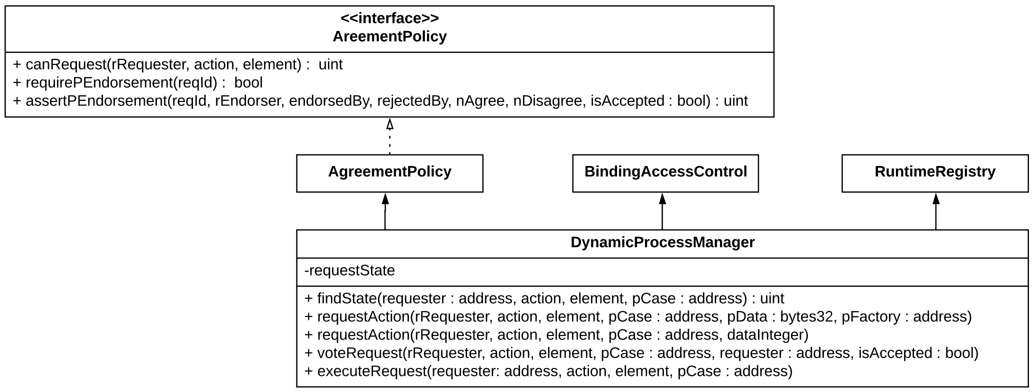

<<interface>>
AreementPolicy
+ canRequest(rRequester, action, element) : uint
+ requirePEndorsement(reqId) : bool
+ assertPEndorsement(reqId, rEndorser, endorsedBy, rejectedBy, nAgree, nDisagree, isAccepted : bool) : uint
AgreementPolicy
BindingAccessControl
RuntimeRegistry
DynamicProcessManager
-requestState
+ findState(requester : address, action, element, pCase : address) : uint
+ requestAction(rRequester, action, element, pCase : address, pData : bytes32, pFactory : address)
+ requestAction(rRequester, action, element, pCase : address, dataInteger)
+ voteRequest(rRequester, action, element, pCase : address, requester : address, isAccepted : bool)
+ executeRequest(requester: address, action, element, pCase : address)

Figure 13: Class diagram of the smart contracts derived from the agreement policies.

Section 4. This contract maps the state of the requests for each process case in a variable called REQUESTSTATE. Here, we follow the same principles as in the BINDINGACCESSCONTROL, thus the DYNAMICPROCESSMANAGER contract is hard-coded, deployed once to the blockchain and, when possible, the data is compressed into bitsets.

The overloaded functions `requestAction` in the DYNAMICPROCESSMANAGER support the different types of data to be updated at runtime, i.e., hashes and addresses of a process to link, and integer indexes related to roles and dynamic gateways. These functions interact with the BINDINGACCESSCONTROL contract to retrieve the roles of the actor starting the requests, whose rights are verified later. Once in the state GRANTED, a request must be explicitly executed by the actor that started it, via the function `executeRequest`, which enforces the action using the data provided in the request, and changes the request state to UNGRANTED (implementing the once-only execution constraint). We avoid the automatic execution of granted requests because, in that case, the actor who voted the last would have to pay the fees incurred by this execution, which should be covered by the requester instead. The function `executeRequest` requires an interaction with other components/contracts, i.e., BINDINGACCESSCONTROL or RUNTIMEREGISTRY [6], to update the control-flow accordingly. The remaining functions `findState` and `voteRequest` follow the same logic as in the BINDINGACCESSCONTROL contract, but concerning the request states.

The agreement policy specification is compiled into the AGREEMENTPOLICY contract. Below we discuss how the compiler generates the function `canRequest`. Generating the function `assertPEndorsement` is similar to the function `assertNEndorsement` in the BindingPolicy contract.

Function `canRequest` consists of nested conditional (`if/else-if`) blocks. The outermost conditional blocks check which action is being triggered (`link-process`, `link-role` or `choose-path`). The second-level blocks check which role is triggering this action. Finally, the third-level conditional blocks check to which control-flow element the action is applied in order to determine whether or not the role in question has the right to trigger the action on this control-flow element according to the policy.

For example, the statement "2 (actor) requests 1 (action) on 3 (control-flow element)" is translated into:

```
function canRequest(uint rRequester, uint action, uint element)
    returns(uint) {
   if (action == 1) {
      if(rRequester == 3) {
         if(element == 2) {
            return requestID;
         } else if /* remaining elements s.t. role 3 can request
             action 1 */
           ...
      } else if /* remaining roles that can request action 1 */
         ...
   } else if /* remaining actions defined by the agreement */
   ...
   return 0;
}
```

6.3. Experimental Setup

We conducted an evaluation to answer the following question: How does the cost (in gas/ether) of enforcing role-binding and agreement policies increase depending on the size and complexity of the policy statements?[7] We decompose this question into three: (Q1) How do the costs of deploying the generated smart contracts vary with the size of the policy? (Q2) How do the costs of executing the runtime operations vary with the size of the policy? (Q3) How does the combined cost of enforcing a process model and policy vary with the size of the policy?

It follows from Section 6.1 that the costs derived from a role-binding policy depend on the number of roles to nominate and the number of conjunction

[7]In Ethereum, gas is linearly related to throughput, see Section 2.1. So by answering this question we also indirectly answer the related throughput question.

sets in the binding/endorsement constraints. Thus, we designed the following experiments. In (E1), we varied the number of nomination statements in a policy from 1 to 40, without any binding or endorsement constraints. (E2): we fixed the number of statements to 40, selected one statement, and gradually increased the size of its conjunction set in binding constraint from 1 to 40. (E3) fixes the number of statements to 40, and gradually added a binding constraint with one conjunction set to each of the 40 statements. (E4,E5): the experiments E3 and E4 were repeated for the endorsement constraint (instead of the binding constraint). For (E6), we generated a policy with 40 roles such that each statement includes a binding constraint stipulating that the nominated actor must belong to the role in the previous statement and that the nomination must be endorsed by all actors nominated in previous statements. (E7): starting from a BPMN model with only one task, we iteratively expanded it, one task at a time (up to 40), and assigned each task to a different role. In addition, from a BPMN model with 40 tasks we iteratively increased the number of roles executing them (up to 40). In this last experiment, once a role was bound to an actor, we checked that the corresponding task could be performed. Note that the evaluation focuses on nomination statements, but the release statements are symmetric.

It also follows from Section 6.1 that the costs derived from an agreement policy depend on the number and structure of the triplets in the statements, i.e., role, action and control-flow element, and the endorsement constraints. Thus, we designed three experiments (E8-E10), each of which increases the number of statements from 1 to 40, to check the cost derived from different possible combinations of triplets (with indexes between 1 and 40) without any endorsement constraint. (E8) fixes actor and action, and ranges the control-flow element from 1 to 40. (E9) fixes action, and gradually increment the pair actor control-flow element from 1 to 40. This experiment is equivalent to fixing action and control-flow element, from the code generation perspective, but grants a full execution of each statement (without rejection) because it avoids the case that once the request is GRANTED, it invalidates the remaining requests to the same control-flow element. (E10) fixes the actor and control-flow elements, and varies the number of actions from 1 to 40. Although this paper focuses on three actions only, experiment E10 illustrates the costs of an eventual extension of the policies with new actions. Note that smart contracts implementing and enforcing agreement policies are independent of the smart contracts derived from the process models. Thus, the experiment randomizes the generation of generic actions,

Table 1: Relationships between experiments and research questions (RQ).

RQ	**Exp.**	**Relationship**
Q1	E1 - E5	Retrieves the deployment costs of smart contracts generated from role-binding policies, illustrating how they vary with the size.
	E8 - E11	Retrieves the deployment costs of smart contracts generated from agreement policies, illustrating how they vary with the size.
Q2	E1 - E6	Executes and retrieves the costs of the operations `nominate` and `vote` in the smart contracts generated from role-binding policies.
	E8 - E11	Executes and retrieves the costs of the operations `request` and `vote` in the smart contracts generated from agreement policies.
Q3	E6	Estimates the upper bound on the deployment costs for role-binding policies and process models up to 40 roles and tasks, respectively.
	E7	Retrieves the deployment costs of the contracts relating policies and process models, and the costs of executing the operation `canPerform`, illustrating how they vary with the size.

i.e., adding a random label to control-flow elements. Then, we can generate the corresponding agreement policy, including mock actions, and perform the operations to make the actions granted. However, we cannot perform `executeRequest` because even when the mock action is in state GRANTED, it is not linked to an existing action to update the control-flow element in the DynamicProcessManager contract. Finally, in (E11), we assess the voting by ratio; to this end, experiment (E8) is repeated but including an endorsement constraint, such that the `i-th` iteration contains a ratio expression where 100% out of `i` roles must accept the request. Here, we did not consider boolean expressions on the endorsement constraints as they were assessed in experiments E4-E5.

Table 1 summarizes how the experiments designed contribute to answering the proposed research questions.

We implemented a replayer in Java that generates the (role-binding and agreement) policies, triggers their compilation and deployment, and executes the runtime operations via CATERPILLAR's REST API. For each transaction included in the blockchain, CATERPILLAR sends some meta-data that includes block number, consumed gas, transaction hash which is collected and assessed by the replayer. For the experimentation we run a *Node.js* based Ethereum client named *ganache-cli*[8] which is widely used to simulate a full client for developing and testing purposes on Ethereum.

6.4. Experimental Results and Discussion

In order to answer the question Q1, deployment costs of the role-binding policies in the experiments E1-E5 are plotted in Fig. 14. It shows that deployment costs increase quasi-linearly with the size and complexity of the policy[9]. The most straightforward role-binding contract (with a single role bound to case-creator) costs 154,167 gas. As expected, the most pronounced growth in cost occurs for endorsement constraints (E4-E5) as they produce more instructions during code generation. We observe an increase of around $16.0 - 19.0\%$ when adding a new endorsement constraint and $5.0 - 6.5\%$ when adding one conjunction set to a constraint. Experiments E2-E3 show that adding a binding constraint increases the cost by $4.0 - 5.7\%$ while adding one conjunction to a constraint adds $2.4 - 3.5\%$ overhead. E1 shows that adding one unrestricted statement to nominate a role adds $4.0 - 4.5\%$ overhead.

Continuing with experimental question Q1, Fig. 15 plots the deployment costs of the agreement policies in experiments E8-E11. Like in the role-binding policies, the deployment costs increase quasi-linearly with the size and complexity of the agreement policy. The most straightforward agreement contract (with a single role proposing one action on a control-flow element) costs 142,293 gas. Like in the role-binding policies, the most pronounced growth in cost occurs for ratio expressions in the endorsement constraints (E11). We observed an increase of 8.0% on average when adding a new ratio expression. As expected, the number of roles allowed to vote in a ratio expression does not affect/increase the deployment costs as they are always encoded in one bit-set, i.e., a single integer number. Accordingly, using ratio expressions, instead of boolean expressions, leads to a reduction in the deployment costs. Besides, ratio expressions are less restrictive as they rely on the amount instead of who is casting the votes. Although endorsement

[8]`https://github.com/trufflesuite/ganache-cli`

[9]Note that figures 14 and 15 displays not the entire cost of the policies, but the growth costs derived from the instructions assessed in each experiment.

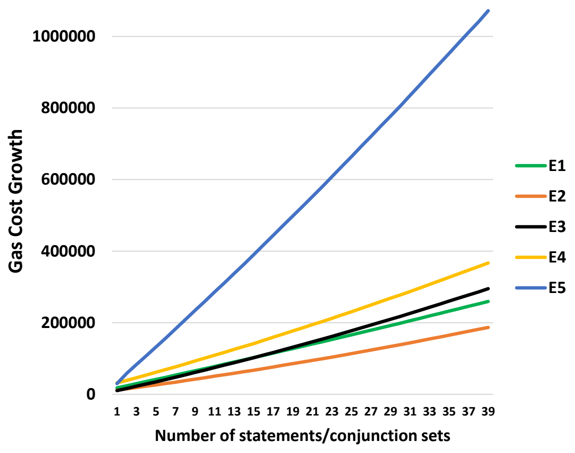


Figure 14: Growth of deployment costs with size of a role-binding policy.

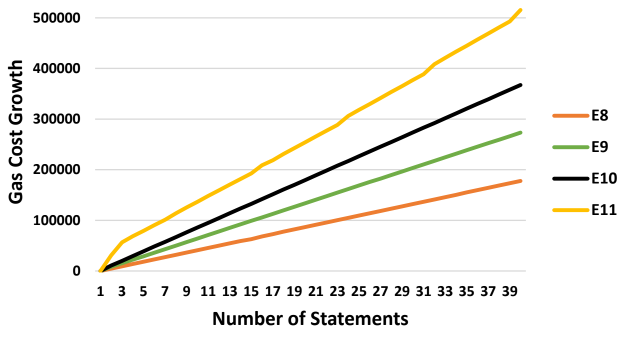


Figure 15: Growth of deployment costs with size of an agreement policy.

constraints with boolean expressions entail a higher cost, they allow finer restrictions regarding the specific sets of roles that are required to achieve a given outcome; i.e., at least one entire set must accept one operation for it to become active.

Finally, regarding question Q1, we observed an increase of about 3.0%, 5.0%, and 7.0% on the deployment costs in experiments E8, E9, and E10, respectively. It shows how agreement policies are less costly when only a role is allowed to perform a single action on a set of control-flow elements. On the contrary, agreement policies cost more if the number of actions increases. The latter is convenient because our proposal focuses only on three actions, i.e., `link-process`, `link-role` and `choose-path`. Overall, we observed that adding a nested condition lead to an increase in the cost of about 2.0%.

Table 2: Cost of the nomination and vote operations on the role-binding policies.

		E1	**E2**	**E3**	**E4**	**E5**
Nom.	Min.	151,586	112,476	111,407	132,417	131,493
	Max.	152,638	152,790	113,447	152,746	153,800
	Ave.	151,948	151,270	112,277	151,738	142,660
Vote	Min.	-	-	-	76,845	77,184
	Max.	-	-	-	78,136	78,184
	Ave.	-	-	-	77,463	77,541

Table 3: Cost of the request and vote operations on the agreement policies.

		E8	**E9**	**E10**	**E11**
Req.	Min.	168,075	168,075	183,049	168,135
	Max.	183,049	183,049	183,049	183,112
	Ave.	169,118	169,118	183,049	169,178
Vote	Min.	-	-	-	50,397
	Max.	-	-	-	81,399
	Ave.	-	-	-	51,889

In order to answer the experimental question Q2, we observed that costs of the runtime operations vary slightly with the number and the order of statements and conjunction sets in the constraints. The cost to nominate a role is higher when the corresponding policy statement is at the end of the policy. Similar behavior exists for binding and endorsement constraints. From the perspective of the algorithmic computational complexity [47], the functions generated from the policies run in constant time. However, the cost variations are due to the specificity of Ethereum in which gas costs are affected by the number of bytecode instructions executed. Hence, in a function with `if-else-if` instructions, the cost increases with the number of evaluated conditions.

Also related to question Q2, Table 2 shows the min, max, and average costs to perform the nominate and vote operations in experiments E1-E5. Similarly, Table 3 shows the costs related to perform the request and vote operations in experiments E8-E11. Note that voting is less costly than nominating, and that nomination costs are lower when restricted by binding constraints compared to endorsement constraints. Requesting an action in an agreement policy is slightly costlier than nominating in a role-binding policy. The latter is an expected result because while nomination statements involve two entities, nominator and nominee, request statements contain three of them, action, role, and control-flow element, which indeed leads to more instructions in the smart contract. Finally, the voting operations derived from ratio expressions (cf. experiment E11) are less costly as they require a more straightforward encoding than voting from boolean sets.

A critical remark when answering the experimental question Q3 constitutes that smart contracts derived from process models, role-binding and agreement policies work independently. In other words, the deployment and execution costs of the smart contract generated from a process model are not directly increased by the policies. However, instead, they depend on the size and structure of the process and the control-flow generation strategy. The overhead introduced by the policies on the process execution comes from the deployment costs of smart contracts derived from the policies. Another source of overhead comes from the execution costs of the operations `canPerform`, whose costs rely on policies and not on the process model. Accordingly, our experimentation mainly focuses on the costs derived from the policies. However, we design the experiments E6 and E7 to approximate an upper bound from adding the deployment and execution costs of the policies aligned to research question Q3. Specifically, assessing the combined cost of executing a process model with an associated role-binding policy (experiments E6 and E7) has several components.

RB1 Deployment of the smart contract BindingAccessControl at a fixed cost of 1,340,098 gas.

RB2 Deployment of the smart contract BindingPolicy generated from the role-binding policy, with costs ranging from 154,167 (simplest with only one role) to 1,803,898 gas (largest with 40 roles in E6).

RB3 Deployment of the smart contract TaskRoleMap to relate roles in the policy to tasks in the process model. In experiment E7, we observed a linear growth in the deployment cost of this contract as the number of relations task-role increased, from 129,539 to 241,114 gas units.

BP4 The costs of executing one nominate operation range from 111,407 to 168,270 gas units, while one `vote` operation costs between 50,397 and 78,184 gas units.

RB5 Verifying the right of an actor to execute one task of the process requires invoking the function `canPerform` in the BindingAccessControl.

This function, in turn, invokes the TaskRoleMap contract to retrieve the task-role relation. The costs of executing the function `canPerform` also grew linearly from 31,693 to 33,066 gas units.

On average, deploying agreement policies is less costly than role-binding policies. However, agreement policies rely on a role-binding policy to verify at runtime whether the actor proposing the action on the control-flow is bound to a role with the right to perform it. In numbers from the experiments E8-E11, the costs of the components derived from agreement policies are the following:

A1 Deployment of the smart contract DynamicProcessManager at a fixed cost of 1,055,851 gas.

A2 Deployment of the smart contract AgreementPolicy generated from the agreement policy, with costs ranging from 142,293 (simplest with only one request) to 674,851 gas (largest with 40 requests). We excluded here the costs derived from endorsement constraints as they are proportional to those in the role-binding policies.

A3 The costs of executing one `propose` operation range from 168,075 to 183,112 gas units, while one `vote` operation costs between 50,397and 78,184 gas units.

A4 The cost of the function `executeRequest` in the DynamicProcessManager contract depends on the process model, as it triggers operations defined as part of the control-flow, e.g., linking a process involves its instantiation and possible execution of enabled elements. Therefore, the costs depend on the control-flow implementation, and not on the structure or size of the agreement policy.

It is essential to consider that the smart contracts derived from the role-binding and agreement policies can be reused (after being deployed only once). In contrast, the contracts handling the process models (i.e., control-flow) typically require a new deployment for each process case. Accordingly, several executions of a process model lead to the amortization of the deployment costs incurred for the policies (if these are reused).

Estimating the overhead added by the policies to the process execution is not straightforward due to the many combinations and scenarios coming

Table 4: Comparison of deployment and execution costs between role-binding policies and business process models; with amortization by reuse (A) and without.

Smart Contract	**Depl.**	**Exec.**	**Depl. (A)**	**Exec. (A)**
BindingAccessControl	1,340,098	343,657	252	343,657
TaskRoleMap	241,114	-	45	-
BindingPolicy (min)	154,167	111,407	28	20
BindingPolicy (max)	1,803,898	67,714,320	339	12,735
Process Model (C)	2,830,063	1,088,315	2,830,063	1,088,315
Process Model (I)	543,503	652,784	543,503	652,78

from the design choices when creating and putting together process models and policies. To that end, we considered a BPMN model and the corresponding event log of a real-world business process, named Invoicing, used in the experiments in [6, 16]. The process model has 60 BPMN elements, and 40 of them involve the interaction of an external actor. The event log contains 5317 traces and 55,260 events.

To estimate the overhead, we collected the deployment and execution costs (without any policy) of the process from the two engines implemented by Caterpillar; i.e., following compiled [6] and interpreted [16] approaches, respectively. We also calculated minimum and maximum deployment and execution bounds for the role-binding policies by combining the data collected in the experiments E1-E11. Specifically, the minimum cost comes from a policy, including a unique role, which in turn executes all the tasks in the process model (no voting required). In contrast, the maximum cost corresponds to a policy with 40 roles (one role per task) in which the nomination of an actor requires the endorsement of all the previously nominated actors. Also, we calculated the costs under two possible scenarios: (i) the policies are deployed and the operations performed for each process case, (ii) they are deployed/executed once and then reused in all the process cases to estimate the amortized costs.

Table 4 illustrates the values in which deployment and execution are expected to range. The letters C and I, following the process models label, correspond to the variants compiled and interpreted, respectively. The total costs[10] without reusing the policies would add an overhead between 1,735,379

[10]The total costs include the fixed cost contracts BindingAccessControl and TaskRoleMap, and the corresponding BindingPolicy (min or max accordingly). Be-

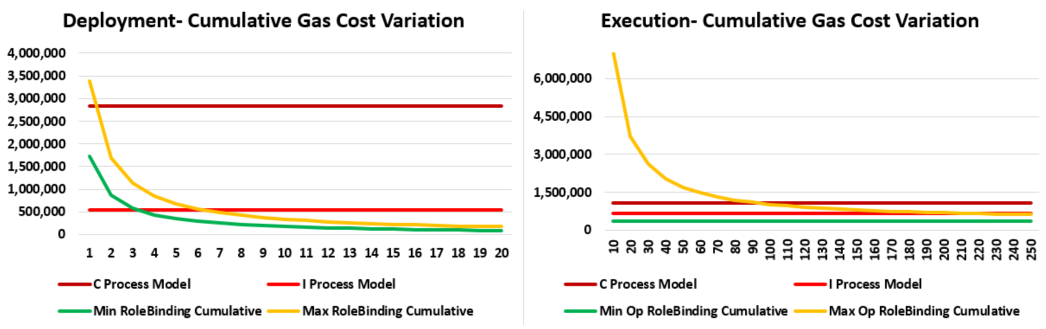


Figure 16: Variation of the amortized deployment and execution costs of role-binding policies by reusing them across different process cases.

to 3,385,110 gas units for deployment, and between 455,064 to 68,057,977 gas units for execution. However, the calculation of the upper bound assumes the extreme scenario with 40 nominations and 780 endorsements in the worst-case scenario. Instead, for example, the traces in the event log of the invoicing process include between 4 and 27 events, meaning that many nomination/voting operations are not required. In contrast, these costs significantly amortize when reusing one single policy with the same actors in all the process cases (labelled with A in Table 4). Then, the total costs range from 325 to 636 gas units for deployment, and from 343,677 to 356,392 gas units for execution. Note that only the operation `canPerform` needs to be executed for each event in each process trace (i.e., to verify the actor rights). The remaining policy operations are not dependent on a specific process case, thus performed only once and reused.

Fig. 16 illustrates how the total deployment and execution costs amortize when reusing the policy in multiple process cases. The deployment costs for the max bound (in yellow) falls below deployment cost of the compiled and interpreted process execution approaches after reusing the policy 2 and 7 times, respectively. The execution is more costly, thus requiring reusability of 91 and 220 times for the costs to fall below of those obtained from the compiled and interpreted approaches, respectively. However, considering the lower bound, the deployment and execution costs are always smaller than the compiled version and falling below the interpreted method after being used

sides, the execution costs of the smart contract TaskRoleMap are included in BindingAccessControl, i.e., from the execution of the function `canPerform`.

4 times. Note that, the numbers offer a rough estimation of the overhead when combining role-binding policies to control the process execution. A similar analysis can be applied to agreement policies. We observed in the experiments that agreement policies are less costly than role-binding policies. Thus, the upper bound for role-binding policies offer a suitable approximation for agreement policies as well.

7. Conclusion

This paper presented an approach to extend blockchain-based collaborative process execution platforms with controlled flexibility mechanisms. Specifically, the contributions of the paper are:

1. A role-binding model and an associated binding policy language that support collaborative binding and unbinding of actors to roles at runtime.

2. An approach for late binding of sub-processes and dynamic selection of execution branches in a process model, together with an associated control-flow agreement policy language, allowing actors to collectively steer the execution of a process instance according to their requirements.

3. A method to verify the consistency of policies defined in the proposed policy specification languages.

4. An approach to compile role binding and control-flow agreement policies into smart contracts for runtime enforcement.

The proposed flexibility mechanisms and associated policy specification languages have been implemented and integrated into the Caterpillar blockchain-based collaborative process execution tool. We evaluated the costs of deploying and executing smart contracts generated from the policy statements on the Ethereum platform. The evaluation shows that the deployment and runtime policy enforcement costs grow linearly with the number of roles, control-flow elements and the complexity of the constraints.

The proposed flexibility mechanisms allow actors in a collaborative process to dynamically adapt the resource and control-flow perspectives of a business process. However, the proposal does not take into account the data perspective. In particular, the proposal does not consider the implications of dynamic role binding and unbinding on the way data is stored and shared between participants. When a participant is bound to a role, in view of performing certain tasks of the process, one expects this participant to have access to the data required to fulfill the role in question. On the other hand, when an actor is unbound from a role, one expects this participant to stop having access to the data associated to this role. A future direction for future work is to develop a role-based data access layer on top of a blockchain platform, which would take into account these interactions between dynamic role binding and data access.

The policies proposed in this paper have been integrated into an approach that compiles process models into smart contracts [6]. This means that, once deployed, the smart contracts that enforce the business process are immutable. Although the proposed approach supports late-bindings and dynamic gateways, other adaptations of the control-flow schema are not allowed. We foresee that the notion of control-flow agreement policy, in conjunction with an interpreted execution of blockchain-based processes [16], may allow us to achieve further flexibility. This is another avenue for future work.

In this paper, we presented a method to detect inconsistencies in role binding policies that may lead to states where an actor cannot be bound to a policy due to circular dependencies between role binding constraints. Other potential inconsistencies in role binding policies may lead to certain parts of a role binding policy being unnecessary. Consider the following example:

```
    A nominates B endorsed-by C
A nominates D in B endorsed-by C, E
```

In this role binding policy, and assuming that there are no multi-instance activities, role D can only be bound to the actor assigned to role B, and hence the endorsement by role E is irrelevant. Another direction of future work is to design verification methods to detect such irrelevant endorsement statements.

While the proposed approach has been designed with the goal of supporting collaborative process execution on blockchain, its field of possible applications is wider. Another future work avenue is to study the applicability of this approach to other blockchain applications where dynamic role binding may be required, e.g., in crowdsourcing and computer-supported collaborative work scenarios.

Acknowledgment

This work was funded by the Estonian Research Council (grant IUT20-55) and the European Regional Development Fund (Dora Plus program).

References

[1] S. Pourmirza, S. Peters, R. M. Dijkman, P. Grefen, A systematic literature review on the architecture of business process management systems, Inf. Syst. 66 (2017) 43–58.

[2] J. Mendling, et al., Blockchains for business process management - challenges and opportunities, ACM Trans. Management Inf. Syst. 9 (2018) 4:1–4:16.

[3] I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, J. Mendling, Untrusted business process monitoring and execution using blockchain, in: Business Process Management - 14th International Conference, BPM 2016, Rio de Janeiro, Brazil, September 18-22, 2016. Proceedings, pp. 329–347.

[4] C. Prybila, S. Schulte, C. Hochreiner, I. Weber, Runtime verification for business processes utilizing the Bitcoin blockchain, Future Generation Computer Systems (2017).

[5] J. Ladleif, M. Weske, I. Weber, Modeling and enforcing blockchain-based choreographies, in: Business Process Management - 17th International Conference, BPM 2019, Vienna, Austria, September 1-6, 2019, Proceedings, pp. 69–85.

[6] O. López-Pintado, L. García-Bañuelos, M. Dumas, I. Weber, A. Ponomarev, Caterpillar: A business process execution engine on the ethereum blockchain, Softw., Pract. Exper. 49 (2019) 1162–1193.

[7] O. López-Pintado, M. Dumas, L. García-Bañuelos, I. Weber, Dynamic role binding in blockchain-based collaborative business processes, in: Advanced Information Systems Engineering - 31st International Conference, CAiSE 2019, Rome, Italy, June 3-7, 2019, Proceedings, pp. 399–414.

[8] X. Xu, I. Weber, M. Staples, Architecture for Blockchain Applications, Springer, 2019.

[9] K. Wüst, A. Gervais, Do you need a blockchain?, in: Crypto Valley Conference on Blockchain Technology, CVCBT 2018, Zug, Switzerland, June 20-22, 2018, pp. 45–54.

[10] Z. Zheng, S. Xie, H. Dai, X. Chen, H. Wang, Blockchain challenges and opportunities: a survey, IJWGS 14 (2018) 352–375.

[11] G. Wood, et al., Ethereum: A secure decentralised generalised transaction ledger, Ethereum project yellow paper 151 (2014) 1–32.

[12] A. B. Tran, Q. Lu, I. Weber, Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management, in: Proceedings of the Dissertation Award, Demonstration, and Industrial Track at BPM 2018 co-located with 16th International Conference on Business Process Management (BPM 2018), Sydney, Australia, September 9-14, 2018., pp. 56–60.

[13] C. Frantz, M. Nowostawski, From institutions to code: Towards automated generation of smart contracts, in: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W), Augsburg, Germany, September 12-16, 2016, pp. 210–215.

[14] Q. Lu, A. B. Tran, I. Weber, H. O'Connor, P. Rimba, X. Xu, M. Staples, L. Zhu, R. Jeffery, Integrated model-driven engineering of blockchain applications for business processes and asset management, arXiv preprint arXiv:2005.12685 (2020).

[15] C. Sturm, J. Scalanczi, S. Schnig, S. Jablonski, A blockchain-based and resource-aware process execution engine, Future Generation Computer Systems 100 (2019) 19 – 34.

[16] O. López-Pintado, M. Dumas, L. García-Bañuelos, I. Weber, Interpreted execution of business process models on blockchain, in: 23nd IEEE International Enterprise Distributed Object Computing Conference, EDOC 2019, Paris, France, October 28-31, 2019, pp. 206–215.

[17] S. Pourmirza, S. Peters, R. M. Dijkman, P. Grefen, A systematic literature review on the architecture of business process management systems, Inf. Syst. 66 (2017) 43–58.

[18] P. Grefen, S. Rinderle-Ma, Dynamism in Inter-Organizational Service Orchestration - An Analysis of the State of the Art, Technical Report, TU Eindhoven, 2016.

[19] C. Cabanillas, M. Resinas, A. del-Río-Ortega, A. R. Cortés, Specification and automated design-time analysis of the business process human resource perspective, Inf. Syst. 52 (2015) 55–82.

[20] M. Zavatteri, C. Combi, R. Posenato, L. Viganò, Weak, strong and dynamic controllability of access-controlled workflows under conditional uncertainty, in: Business Process Management - 15th International Conference, BPM 2017, Barcelona, Spain, September 10-15, 2017, Proceedings, pp. 235–251.

[21] T. Andrews, et al., BPEL4WS, Business Process Execution Language for Web Services Version 1.1, IBM, 2003.

[22] M. Kloppmann, et al., WS-BPEL extension for people - BPEL4People, Joint white paper, IBM and SAP (2005).

[23] C. Pautasso, G. Alonso, Flexible binding for reusable composition of web services, in: Software Composition, 4th International Workshop, SC 2005, Edinburgh, UK, April 9, 2005, Revised Selected Papers, pp. 151–166.

[24] Y. Lu, L. Zhang, J. Sun, Task-activity based access control for process collaboration environments, Computers in Industry 60 (2009) 403–415.

[25] P. Robinson, F. Kerschbaum, A. Schaad, From business process choreography to authorization policies, in: Data and Applications Security XX, 20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security, Sophia Antipolis, France, July 31-August 2, 2006, Proceedings, pp. 297–309.

[26] G. Decker, O. Kopp, F. Leymann, M. Weske, Bpel4chor: Extending BPEL for modeling choreographies, in: 2007 IEEE International Conference on Web Services (ICWS 2007), July 9-13, 2007, Salt Lake City, Utah, USA, pp. 296–303.

[27] J. Wainer, A. Kumar, P. Barthelmess, DW-RBAC: A formal security model of delegation and revocation in workflow systems, Inf. Syst. 32 (2007) 365–384.

[28] L. Bussard, A. Nano, U. Pinsdorf, Delegation of access rights in multi-domain service compositions, Identity in the Information Society 2 (2009) 137–154.

[29] M. Reichert, B. Weber, Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies, Springer, 2012.

[30] R. Cognini, F. Corradini, S. Gnesi, A. Polini, B. Re, Business process flexibility - a systematic literature review with a software systems perspective, Information Systems Frontiers 20 (2018) 343–371.

[31] V. T. Nunes, F. M. Santoro, C. M. L. Werner, C. G. Ralha, Real-time process adaptation: A context-aware replanning approach, IEEE Trans. Systems, Man, and Cybernetics: Systems 48 (2018) 99–118.

[32] N. R. T. P. van Beest, E. Kaldeli, P. Bulanov, J. C. Wortmann, A. Lazovik, Automated runtime repair of business processes, Inf. Syst. 39 (2014) 45–79.

[33] B. Heinrich, M. Klier, S. Zimmermann, Automated planning of process models: Design of a novel approach to construct exclusive choices, Decision Support Systems 78 (2015) 1–14.

[34] J. Klingemann, Controlled flexibility in workflow management, in: Advanced Information Systems Engineering, 12th International Conference CAiSE 2000, Stockholm, Sweden, June 5-9, 2000, Proceedings, pp. 126–141.

[35] M. Adams, A. H. M. ter Hofstede, D. Edmond, W. M. P. van der Aalst, Worklets: A service-oriented implementation of dynamic flexibility in workflows, in: On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, OTM Confederated International Conferences, CoopIS, DOA, GADA, and ODBASE 2006, Montpellier, France, October 29 - November 3, 2006. Proceedings, Part I, pp. 291–308.

[36] A. Marrella, A. Russo, M. Mecella, Planlets: Automatically recovering dynamic processes in YAWL, in: On the Move to Meaningful Internet Systems: OTM 2012, Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2012, Rome, Italy, September 10-14, 2012. Proceedings, Part I, pp. 268–286.

[37] H. Schonenberg, R. Mans, N. Russell, N. Mulyar, W. M. P. van der Aalst, Towards a taxonomy of process flexibility, in: Proceedings of the Forum at the CAiSE'08 Conference, Montpellier, France, June 18-20, 2008, pp. 81–84.

[38] S. W. Sadiq, W. Sadiq, M. E. Orlowska, Pockets of flexibility in workflow specification, in: Conceptual Modeling - ER 2001, 20th International Conference on Conceptual Modeling, Yokohama, Japan, November 27-30, 2001, Proceedings, pp. 513–526.

[39] N. Russell, W. M. P. van der Aalst, A. H. M. ter Hofstede, D. Edmond, Workflow resource patterns: Identification, representation and tool support, in: Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings, pp. 216–232.

[40] Object Management Group, Business Process Model and Notation, version 2.0, 2011. Online at: http://www.omg.org/spec/BPMN/2.0/.

[41] M. Adams, A. H. M. ter Hofstede, W. M. P. van der Aalst, D. Edmond, Dynamic, extensible and context-aware exception handling for workflows, in: On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, OTM Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part I, pp. 95–112.

[42] S. W. Sadiq, M. E. Orlowska, W. Sadiq, Specification and validation of process constraints for flexible workflows, Inf. Syst. 30 (2005) 349–378.

[43] H. Schonenberg, R. Mans, N. Russell, N. Mulyar, W. M. P. van der Aalst, Process flexibility: A survey of contemporary approaches, in: Advances in Enterprise Engineering I, 4th International Workshop CIAO! and 4th International Workshop EOMAS, held at CAiSE 2008, Montpellier, France, June 16-17, 2008. Proceedings, pp. 16–30.

[44] B. Weber, S. W. Sadiq, M. Reichert, Beyond rigidity - dynamic process lifecycle support, Comput. Sci. Res. Dev. 23 (2009) 47–65.

[45] W. M. P. van der Aalst, Business process management: a comprehensive survey, ISRN Software Engineering 2013 (2013).

[46] T. Murata, Petri nets: Properties, analysis and applications., Proceedings of the IEEE 77 (1989) 541–580.

[47] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd Edition, MIT Press, 2009.