DISSERTATIONES
INFORMATICAE
UNIVERSITATIS
TARTUENSIS
20

ORLENYS LÓPEZ PINTADO

Collaborative Business Process Execution on the Blockchain: The Caterpillar System

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

20

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS
20

ORLENYS LÓPEZ PINTADO

Collaborative Business Process Execution on the Blockchain: The Caterpillar System

UNIVERSITY OF TARTU
Press

Institute of Computer Science, Faculty of Science and Technology, University of Tartu, Estonia.

Dissertation has been accepted for the commencement of the degree of Doctor of Philosophy (PhD) in informatics on August 12, 2020 by the Council of the Institute of Computer Science, University of Tartu.

Supervisors

Prof. Marlon Dumas
University of Tartu
Estonia

Prof. Luciano García Bañuelos
Tecnologico de Monterrey
Mexico

Opponents

Prof. Dr. Dimka Karastoyanova
University of Groningen
The Netherlands

Prof. Pierluigi Plebani, Ph.D.
Politecnico di Milano
Italy

The public defense will take place on September 25, 2020 at 12:15 at Delta Building, Narva mnt 18, Room 1021.

The publication of this dissertation was financed by the Institute of Computer Science, University of Tartu.

ISSN 2613-5906
ISBN 978-9949-03-434-5 (print)
ISBN 978-9949-03-435-2 (pdf)

University of Tartu Press
http://www.tyk.ee/

To my parents Olga and Orestes

ABSTRACT

Nowadays, organizations are pressed to collaborate in order to take advantage of their complementary capabilities and to provide best-of-breed products and services to their customers. To do so, organizations need to manage business processes that span beyond their organizational boundaries. Such processes are called collaborative business processes.

One of the main roadblocks to implementing collaborative business processes is the lack of trust between the participants. In the past decade, blockchain technology has emerged as a generic solution to enable a set of parties to collaborate in the absence of mutual trust. Blockchain technology allows a set of parties to maintain an immutable, distributed ledger of transactions and to deploy programs, called smart contracts, that are executed whenever certain transactions occur. These features can be used as essential building blocks for running collaborative business processes between mutually untrusted parties. However, implementing business processes using the low-level primitives provided by blockchain platforms is cumbersome and error-prone. In contrast, established Business Process Management Systems (BPMSs), such as those based on the standard Business Process Model and Notation (BPMN), provide convenient abstractions for rapid development of process-oriented applications.

This thesis addresses the problem of automating the execution of collaborative business processes on top of blockchain technology in a way that takes advantage of the trust-enhancing capabilities of this technology while offering the development convenience of traditional BPMSs. The thesis also addresses the question of how to support scenarios where new parties may be onboarded at runtime, and parties need to have the flexibility to change the default routing logic of the business process, while at the same time ensuring that the execution of the process abides by its basic specification.

The thesis formulates a set of principles and requirements for executing collaborative business processes on the blockchain. It then proposes and evaluates architectural approaches and process modelling concepts to fulfil these principles and requirements. These architectural approaches and modelling concepts are embodied in a novel blockchain-based BPMS that we named CATERPILLAR. Like any process execution engine, CATERPILLAR supports the creation of instances of a process model and allows users to monitor the state of process instances and to execute tasks thereof. The specificity of CATERPILLAR is that the state of each process instance is maintained on the (Ethereum) blockchain and the workflow routing is performed by smart contracts.

The CATERPILLAR system supports two approaches to implement, execute and monitor blockchain-based processes: a compiled approach and an interpreted approach. The compiled approach relies on a compiler from the BPMN modelling notation to the Solidity programming language. The compiler supports an extensive array of BPMN constructs, including sub-processes, multi-instance activities

and event handlers. It also supports processes enhanced with data constraints, which guide the process execution.

This compiled approach takes full advantage of the immutability properties of blockchain platforms: Once deployed, the business logic of a process cannot be altered. On the other hand, this approach is not suitable in dynamic collaboration scenarios where flexibility is a requirement. To handle dynamic collaboration scenarios, CATERPILLAR also supports an interpreted approach to business process execution. It relies on an interpreter of BPMN models, based on dynamic data structures, that is embedded in a business process execution system with a modular multi-layered architecture, supporting the creation, execution, monitoring and dynamic update of process instances. For efficiency purposes, the interpreter relies on compact bitmap-based encodings of process models. An experimental evaluation shows that the proposed interpreted approach achieves comparable or lower costs relative to existing compiled solutions.

Although flexibility is a desirable property, it needs to be restricted in order to avoid participants from steering the process in a direction that is detrimental to others. In order to address this concern, this thesis proposes two models for controlled flexibility in collaborative processes. First, the thesis 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. Second, the thesis 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 thesis 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.

CONTENTS

List of Figures **11**

List of Tables **13**

List of Abbreviations **14**

1. Introduction **15**
1.1. Problem Area: Blockchain-based Business Process Management Systems 15
1.1.1. Collaborative Business Processes 16
1.1.2. Business Process Execution: Compiled versus Interpreted . 17
1.1.3. Access Control, Flexibility and Dynamic Process Execution 19
1.2. Problem Statement 20
1.3. Overview of the Contributions and Outline of the Thesis 25

2. Background **30**
2.1. Business Process Management 30
2.2. Business Process Model and Notation 30
2.3. Blockchain Technology 36
2.3.1. Types of Blockchains and Consensus Protocols 36
2.3.2. Ethereum Blockchain and Smart Contracts 38

3. State of the Art **41**
3.1. Architectures of Business Process Management Systems 41
3.2. Blockchain-Based Collaborative Business Processes: Implementation and Execution 43
3.3. Flexibility in Collaborative Processes 46
3.3.1. Resource Perspective: Access Control, Binding and Delegation Models 46
3.3.2. Control-flow Perspective: Variability, Adaptation, Evolution and Looseness 48

4. Caterpillar: A Blockchain-based Business Process Execution Engine **51**
4.1. Running Example 51
4.2. Architecture of the Caterpillar System 53
4.2.1. On-Chain Runtime and Storage 54
4.2.2. Off-chain Runtime 59
4.2.3. Web Portal 65
4.3. Compiling BPMN into Solidity Smart Contracts 66
4.3.1. Process variables and external resources 66
4.3.2. Control-flow Perspective 70
4.3.3. Sub-processes and Reusable Elements 75

4.3.4. Event Handling 78
4.4. Implementation and Evaluation 82
4.4.1. REST API 82
4.4.2. Experimental Setup 85
4.4.3. Experimental Results and Discussion 87
4.5. Summary 89

5. Interpreted Execution of Blockchain-Based Business Process Models 90
5.1. Extending the Architecture of the Caterpillar System with the Interpretation-based Engine 90
5.1.1. On-Chain and Storage Layer 92
5.1.2. Off-Chain Access and Process-Aware Layers 94
5.2. Control-Flow and Data Representation 98
5.3. BPMN Interpreter Operation 101
5.4. Implementation and Evaluation 106
5.5. Summary 109

6. Controlled Flexibility in Blockchain-Based Business Processes 110
6.1. Dynamic Role Binding 110
6.1.1. Binding Policy Specification Language 111
6.1.2. Runtime Role-Binding Operations 113
6.2. Control-Flow Flexibility and Agreement Policies 116
6.2.1. Agreement Policies on Control-Flow 117
6.2.2. Runtime Agreement Operations 119
6.3. Policy Consistency Verification 120
6.4. Implementation and Evaluation 124
6.4.1. Compiling Role-Binding Policies into Smart Contracts . . . 125
6.4.2. Compiling Agreement Policies into Smart Contracts 128
6.4.3. Experimental Setup 129
6.4.4. Experimental Results and Discussion 131
6.5. Summary 137

7. Conclusion and Future Work 138
7.1. Summary of contributions 138
7.2. Future work 139

Bibliography 141

Appendix A. Code Repositories 160

Acknowledgement 161

Summary in Estonian 162

Curriculum Vitae 164

Elulookirjeldus (Curriculum Vitae in Estonian) **165**

List of original publications **166**

LIST OF FIGURES

1. Relations between tasks, roles, blockchain accounts, and actors (blockchain case vs. conventional case). 19
2. Overview of the process execution on CATERPILLAR 27
3. Example of the activities supported by CATERPILLAR. 31
4. Events supported by CATERPILLAR: (a) Start top-level process, (b) Start event-sub-process interrupting, (c) Start event-sub-process non-interrupting, (d) Intermediate catching, (e) Intermediate boundary interrupting, (f) Intermediate boundary non-interrupting, (g) Intermediate throwing, (h) End. 33
5. Example of the gateways and flow as supported by CATERPILLAR. 35
6. Blockchain: Blocks and Transactions (adapted from [181]). 36
7. Untrusted peer-to-peer network representing the blockchain. . . . 37
8. Basic architecture of BPMSs (from [45]). 41
9. Running example: An order-to-cash process (1), with a shipment sub-process (2) 52
10. The architecture of CATERPILLAR: the compilation-based engine. 55
11. Interfaces with their operations in the smart contracts managed by the compilation-based engine of CATERPILLAR. 57
12. Caterpillar's compilation process 60
13. Process Instantiation through Caterpillar's compilation-based engine. 62
14. BPMN elements supported by CATERPILLAR. 66
15. Life-cycle of BPMN elements in CATERPILLAR: (4) external/reusable (3) internal 71
16. Nested subprocesses with propagation of error events. 76
17. Simple BPMN model. 91
18. Extended architecture of the CATERPILLAR system: the interpretation-based engine. 92
19. Graphical representation of the the control-flow and data perspectives smart contracts deployed to execute two cases of the process in Figure 17. 94
20. Parsing of BPMN models on the CATERPILLAR's interpretation-based engine 96
21. Process set-up and instantiation on the CATERPILLAR's interpretation-based engine. 97
22. Bit associated to each element/characteristic when encoding the element description as `typeInfo`. 99
23. Running example: (1) An *Order-to-cash* process linked, via call activities, to two reusable sub-processes; (2) *Shipment* and (3) *Invoicing*. 111
24. BNF grammar describing the basic statement syntax of a binding policy. 112

25. Life-cycle of a role within a case. 114
26. A more flexible variant of the sub-process GOODS SHIPMENT displayed in Figure 23. 116
27. BNF grammar describing the basic statement syntax of an agreement policy. 117
28. Life-cycle of an action to be performed at runtime. 119
29. Sample binding policy 122
30. Symbolic representation of the binding policy in Figure 29 122
31. Nomination net for binding policy in Figure 29 122
32. Net encoding condition $(A \wedge B) \vee (B \wedge C)$ 123
33. Binding policy with circular dependency and its nomination net . . 123
34. Class diagram of the smart contracts derived from the role-binding policies. 126
35. Class diagram of the smart contracts derived from the agreement policies. 128
36. Growth of deployment costs with size of a role-binding policy. . . 132
37. Growth of deployment costs with size of an agreement policy. . . . 132
38. Variation of the amortized deployment and execution costs of role-binding policies by reusing them across different process cases. . . 137

LIST OF TABLES

1. CATERPILLAR's compilation-based engine REST API. 83
2. Datasets used in the evaluation. 86
3. CATERPILLAR's compilation-based engine: process instantiation and execution costs. 88
4. CATERPILLAR's interpretation-based engine REST API. 107
5. CATERPILLAR's interpretation-based engine: setting-up costs. . . 107
6. CATERPILLAR's interpretation-based engine: process instantiation and execution costs. 108
7. Relationships between experiments and research questions (RQ). . 131
8. Cost of the nomination and vote operations on the role-binding policies. 133
9. Cost of the request and vote operations on the agreement policies. . 133
10. Comparison of deployment and execution costs between role-binding policies and business process models. 136

LIST OF ABBREVIATIONS

ABI	Application Binary Interface 56–58
API	Application Programming Interface . . 36, 66, 79, 80, 83, 100, 101, 123
BNF	Backus Naur form 105, 110
BPM	Business Process Management 27, 38, 41
BPMN	Business Process Model and Notation . . 16, 19–23, 25, 27, 28, 30, 31, 40–43, 47, 49, 51, 52, 54–56, 59–61, 63, 65, 66, 68, 69, 71, 72, 74, 79, 80, 82, 83, 85–87, 89–93, 95, 100, 101, 103, 104, 110, 111, 117, 122, 128
BPMS	Business Process Management System . . 15, 16, 19–23, 27, 28, 38, 39, 41, 42, 47, 83, 85, 128
CIO	Chief Information Officer 15
dapp	Decentralized Application 24, 35
DMN	Decision Model and Notation 41
ETH	Ether 125, 126
EVM	Ethereum Virtual Machine 35, 36, 56, 69
HTTP	HyperText Transfer Protocol 80–82
IoT	Internet of Things 19, 43, 91
IPFS	InterPlanetary File System 51, 52
JSON	JavaScript Object Notation 56, 80–83
JSON-RPC	Remote Procedure Call protocol encoded in JSON 36
MDE	Model-driven Engineering 40, 41
RBAC	Role-Based Access Control 19, 43
REST	Representational State Transfer . 39, 79, 80, 83, 100, 101, 117, 123
RPC	Remote Procedure Call 36
SOA	Service-Oriented Architectures 38
URI	Uniform Resource Identifier 80, 101
URL	Uniform Resource Locator 80–82
WfMC	Workflow Management Coalition 38
WS-BPEL	Web Service Business Process Execution Language 39, 44
XML	Extensible Markup Language 80

1. INTRODUCTION

Business processes are a core asset of organizations. They integrate systems, data, and resources to accomplish organizational goals by delivering a service or product to a client [45]. While traditional intra-organizational processes focus on one organization, collaborative inter-organizational processes, on the other hand, span multiple organizations. Nowadays, increasing pressures on organizations to be competitive and comply with the growing demands coming from globalization have heightened the importance of collaborative processes. However, lack of trust among organizations is a significant roadblock to implementing and executing collaborative processes, which typically leads to companies relying on trusted third parties to serve as mediators [34].

Blockchain technology allows mutually untrusted parties to execute collaborative business processes without relying on a central authority [99]. Specifically, blockchain platforms allow the parties in a collaborative business process to record the state of the process on a tamper-proof and decentralized ledger, which also stores and executes programs (called smart contracts) that implement transactions on top of the ledger. The combination of a tamper-proof and decentralized ledger with smart contracts provides the basic building blocks to implement collaborative (inter-organizational) business processes involving mutually untrusting parties [99, 163]. Not in vain, several existing blockchain applications implement business processes involving multiple independent participants, such as supply chain management processes [75, 146, 152].

This thesis addresses the problem of automating the execution of collaborative business processes, with emphasis on exploiting blockchain capabilities to provide a tamper-proof execution under the dynamic scenarios existing in collaborative processes. The specific problem areas addressed in this thesis are described in Section 1.1. Next, Section 1.2 introduces the scope, research questions, and principles that guide the construction of CATERPILLAR, a prototype of a business process management system that we created to demonstrate the execution of blockchain-based collaborative processes. Finally, Section 1.3 provides the contributions and outline of the thesis.

1.1. Problem Area: Blockchain-based Business Process Management Systems

Implementing collaborative business processes using the low-level primitives provided by blockchain platforms is cumbersome, error-prone, and requires specialized skills. According to a survey by Gartner [129], around one-fifth of relevant surveyed Chief Information Officers (CIOs) stated that one of the major roadblocks for the adoption of blockchain technology in their companies is that it requires highly specialized teams that are difficult to put together. In contrast, established Business Process Management Systems (BPMSs) provide convenient ab-

stractions for the rapid implementation of intra-organizational business processes by taking as a starting point a business process model represented, for example, in the Business Process Model and Notation (BPMN) [58] standard. These abstractions make it possible to implement and maintain process-oriented applications based on process models without requiring low-level or specialized development skills.

Business Process Management Systems (BPMSs) are information systems that automate the execution of business processes typically described by a model. They coordinate the interactions of human actors with machine operations, providing organizations with a flexible way to manage their processes [155]. Nowadays, a wide variety of BPMSs cover some of the traditional steps of the business process life-cycle, i.e., identification, discovery, analysis, redesign, implementation, execution, monitoring, adaptation and evolution [45]. However, typical issues like interoperability, dynamic interactions, trust, and security are frequently ignored or not adequately addressed in inter-organizational collaborations between mutually untrusted parties; e.g., a recent survey found that only 30% of the BPMSs analyzed handle collaborative processes [123].

1.1.1. Collaborative Business Processes

Several challenges exist in the field of collaborative business processes. One comes from the autonomy of organizations and a lack of trust, e.g., which partner performs a given task, what data each participant can access, and so forth. Besides, the visibility of a party regarding the activities performed by the other participants is often limited. Although some agreements are possible, choosing a reliable enforcement mechanism or a mediator can be a problem even more significant than the process automation itself [123]. Another challenge is to have the flexibility to allow changes in the process requirements in real time [122, 128], such as dynamic onboarding and replacement of business parties. In the logistics field, some collaborative processes require dynamic binding and re-binding. For example, in a buyer-supplier-carrier process, the carrier might sometimes be appointed (selected) by the supplier and other times by the buyer. Also, sometimes the seller might have the right to change the carrier after the initial appointment, e.g., if the initially appointed carrier is not able to pick up the merchandise on time. In other words, the responsibility for a given task can change across cases.

Blockchain provides a suitable platform to execute collaborative processes in a trusted manner [69, 152]. Overall, blockchains offer a transparent and tamper-proof platform which allows parties to track the actions of the others. Two main concepts of blockchain technology support the above. First, no central authority is required, but instead, a peer-to-peer network stores a copy of the information as a linked sequence of blocks, and each block contains a set of valid transactions. Second, the inclusion of smart contracts provides support to implement business rules and update the process status as programmable scripts [12]. By executing

collaborative processes as smart contracts, blockchain will serve as an immutable and trusted ledger that records and validates every operation during the process execution. Transactions can be restricted to a set of roles; i.e., they can be executed only by a participant with an authorized signature. Participants may access the transactions generated by the process, which enables the monitoring of its execution [35, 101]. Besides, with encryption, it is possible to restrict access to data that is public but only readable by participants with the corresponding privilege [99].

Blockchain offers a transparent and cryptographically-secured environment built on consensus within a distributed peer-to-peer network. Such a closed and complex ecosystem opens a broad range of opportunities to perform collaborative processes, but it also introduces some crucial challenges. Among those, the amount of computational power and data storage capabilities are limited. Thus, when implementing blockchain-based processes, a critical decision relies on which data/operations must be stored/performed in the blockchain (i.e., on-chain for short) and which outside of the blockchain (i.e., off-chain) [46, 172, 174]. As such, several operations on collaborative processes rely on data stored off-chain. However, there is no guarantee about the integrity of off-chain transactions, e.g., they can be compromised in the presence of unknown or untrusted sources. Therefore, the participants would need to collaboratively assert the off-chain decisions before proceeding with the execution of the process on-chain.

1.1.2. Business Process Execution: Compiled versus Interpreted

Existing approaches to blockchain-based business process execution are mostly based on the idea of compiling each process model into a set of smart contracts [99]. Once deployed on a blockchain platform, these smart contracts control the instantiation of the process as well as every change to the state of a process instance [99, 163]. In this way, this approach ensures that every process execution abides by its corresponding process model.

In these compiled approaches, the deployed smart contracts are model-dependent. More specifically, each element in the process model is statically encoded as a set of instructions embedded into the smart contract. For example, consider a simple process model with one start event followed by a task whose execution enables an end event. In a compiled approach, the code of the derived smart contract would contain three blocks of fixed instructions, i.e., implementing the execution logic of each element. This approach exploits immutability as a source of trust. Once deployed in the blockchain, the code of the smart contract is immutable. Thus no participant can modify the process execution in their benefit. However, it comes at the cost of inflexibility. Even a minor change to the process model requires a full recompilation of the model and the deployment of a new set of smart contracts. Subsequently, new instances of the process are created using the new set of smart contracts, while pre-existing process instances remain tied to the old version of the model. This lack of flexibility is problematic in the

context of business processes that are subject to constant evolution as well as processes with a large number of pathways and exceptions, which typically cannot be captured upfront via a single process model [128].

In addition to a lack of flexibility, another drawback of compiled approaches is the inefficiency induced by deployment costs. Indeed, each model (or version thereof) is encoded via separate smart contracts, and in contemporary public blockchain platforms, such as Ethereum,[1] every smart contract deployment entails costs proportional to the contract's size. Similarly, compiled approaches are often redundant as they repeat instructions in the smart contracts derived from different models sharing a similar structure, and duplicating code of elements of the same type. For example, a process model with two script tasks may lead to two similar blocks of instructions to the fact that both activities share the same execution logic. However, these efficiency limitations mainly apply to blockchain platforms in which participants must pay for deploying a transaction. In blockchain platforms not tied to cryptocurrencies such as Hyperledger [27], deploying the smart contracts incurs no fee. Besides, the increase in the size of the contract due to repeated instructions is not a significant issue.

In contrast, an interpreted execution of processes relying on data can prevent the issues related to the flexibility and deployment costs. For example, instead of being encoded as fixed blocks of instructions, the information about each element in a process model can be stored in a generic data structure. Indeed, each process perspective, i.e., control-flow, data and resource, would require different data structures. Then, participants may add, update or remove elements at any time during the process execution, supporting a more flexible implementation.

A singleton smart contract (the interpreter) encodes the semantics of the modelling language from the corresponding process data and is capable of executing a given process model. Hence, the combination of the interpreter with the data structures approach reduces the deployment costs, i.e., an update on the process model would require a local update of the data structure and not a redeployment of the entire smart contract. Note that, contrary to traditional programming languages in which interpreted solutions are less efficient, in public blockchain platforms, the size of the code and deployment costs play a predominant role in the total costs of executing smart contracts, and thus efficiency largely depends on these parameters. In this setting, an interpreted approach reduces both the size of the code and the deployment costs compared to compiled solutions.

However, the flexibility of interpreted approaches may lead to trust issues or inconsistencies on the process during the execution. For example, if a participant tries to take advantage of updating the process at runtime (not possible in compiled approaches), the other participants require a more sophisticated control mechanism to prevent that behaviour. Besides, adding, updating or removing elements at runtime may lead to deadlocks if it is not performed correctly. None of

[1] https://ethereum.org/

these two issues above is present in compiled approaches.

In summary, compiled approaches are more suitable in scenarios in which flexibility is not a requirement, or in blockchain platforms in which the participants pay no fee for the deployment of the contracts. In contrast, interpreted solutions fit better for processes which are subject to changes at runtime, or to be executed on public blockchains in which the participants must pay a transaction fee for the deployment.

1.1.3. Access Control, Flexibility and Dynamic Process Execution

Access control is an essential aspect of the design and execution of business processes. Mainstream BPMSs rely on static Role-Based Access Control (RBAC) models. In these models, any worker who plays a role is allowed to perform any task associated with this role in any instance of the process, plus additional constraints such as separation of duties [16, 136]. However, this approach is unsuitable for collaborative processes involving untrusted actors.

The characteristics of blockchain technology shift the role-binding problem in two ways. First, rather than groups or individual users being bound to roles, it is required to link blockchain accounts (or identities) to roles, as shown in Figure 1. These accounts, in turn, are controlled by users, groups, systems, or Internet of Things (IoT) devices. Second, and more significantly, in open blockchain networks, instances of a collaborative process are created by different actors, and each of these actors trusts one subset of actors but not others.

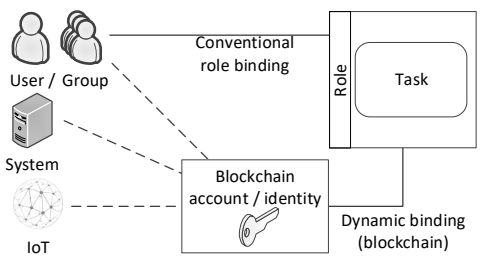


Figure 1: Relations between tasks, roles, blockchain accounts, and actors (blockchain case vs. conventional case).

Existing blockchain approaches execute and monitor collaborative processes using high-level process specifications, such as process models captured in BPMN [82, 125, 148, 163]. These approaches, however, suffer from two limitations that 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. In other words, they commonly do not support access control or adopt a static role-binding approach wherein all actors are bound to roles upon process instantiation.
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 to fit their collective requirements.

To illustrate the need for the first of the flexibility features described above, 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. However, later, the buyer may lose this trust (e.g., if the carrier misses a deadline). After that, 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 binding of actors to roles (the buyer and supplier both need to agree on the carrier).

Meanwhile, to illustrate the need for the second flexibility feature, we consider a case in which some buyers require that a carrier uses a specific type of customs declaration system. Other buyers, however, need a different method. 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 joint 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.

1.2. Problem Statement

This thesis describes CATERPILLAR, an open-source BPMS designed from the ground up to combine the development convenience of a BPMS with the tamper-proof design of a blockchain platform. Like most contemporary BPMSs, CATERPILLAR supports the creation of instances of a BPMN process model and allows managers and process workers to track the state of process instances and to execute tasks thereof. To the best of our knowledge, CATERPILLAR is the first proto-

type to demonstrate how a full-fledged business process execution engine can be deployed entirely on a blockchain platform. Specifically, no off-chain component is required to execute and monitor instances of a process after its deployment.

On the construction of CATERPILLAR, this thesis addresses the problems described by the following research questions:

RQ1 How can the high-level abstractions of BPMSs be combined with the capabilities of blockchain technology to support the execution of collaborative business processes between mutually untrusted parties?

RQ2 How can collaborative processes involving mutually untrusted parties be flexibly and cost-efficiently executed on a blockchain platform?

RQ3 Which access control mechanisms would allow us to capture the wide range of dynamic binding and rebinding scenarios found for collaborative processes between mutually untrusted parties?

In the context of collaborative business processes, blockchain technology allows us to ensure that the parties involved in a collaboration comply with an agreed-upon collaborative process model. Besides, with CATERPILLAR, we aim to reduce the effort of the process participants on their interactions with the blockchain. For example, in a collaborative process involving a purchasing company, a supplier, and a carrier, blockchain technology allows us to ensure that the carrier does not submit the invoice to the supplier before the purchasing company has acknowledged the delivery. To that end, both the supplier and the carrier needs to agree in a BPMN model describing the tasks that they (as well as other possible participants) can perform on the process. Then, one of the participants should submit the corresponding model through the user interface of CATERPILLAR which in turn produces (and later deploys) a set of smart contracts encoding the execution logic of the process. Note that, none of the participants needs to run a blockchain node. Instead, they will connect/interact with a blockchain network through CATERPILLAR, i.e., participants can also decide to which network they want to deploy the contracts. Once deployed, the smart contracts would enforce the process execution as defined in the model. Indeed, participants can verify at any time the state of the process, and execute their tasks only when enabled in the control flow. Also, on the execution of a task, a party can retrieve and send data to the blockchain, i.e., check-out and check-in operations from user tasks in the BPMN model [45]. Note that, participants can directly interact with contracts deployed by CATERPILLAR in the blockchain, i.e., they do not need to use the user interface of CATERPILLAR. Also, they can implement their off-chain components to handle the blockchain interactions if they prefer.

On the design of CATERPILLAR, we considered how compliance concerning a (collaborative) process model could be ensured using one of two approaches [138]. The first approach is compliance by monitoring, which means that the parties record their transactions on a blockchain so that all other parties can check that the process has been executed as agreed. When a party deviates

regarding the agreed-upon process, other parties can detect it and trigger a conflict resolution procedure. This approach is taken by several commercial BPMSs such as Bizagi[2] and Camunda,[3] which offer adapters to record the transactions produced by a process in a blockchain. The second approach is compliance by design. In this latter approach, the parties execute each step of the business process by invoking a transaction on the blockchain. When a transaction is invoked, the blockchain platform checks the current state of the process and the inputs/outputs of the transaction. The transaction is accepted if and only if it complies with the process model. This approach requires that the full specification of the collaborative process is encoded as smart contracts running on the blockchain. CATERPILLAR is an embodiment of this compliance by design approach.

The compliance by design approach is suitable when the level of trust between parties is low, the impact of non-compliance is high, and the cost of conflict resolution is high (e.g., if the parties are in multiple jurisdictions). The latter is the scenario addressed by CATERPILLAR. Conversely, when the parties have some minimum level of trust, the impact of non-compliance is limited, and conflict resolution is straightforward, then an approach based on compliance by monitoring is more suitable. Hence a BPMS that uses the blockchain purely as a secure logging mechanism is sufficient.

In the blockchain setting, BPMN collaboration diagrams could be used with message exchanges between parties (represented as pools). That would be suitable if blockchain was primarily used in its function as a data store (for messages) and communication mechanism (transaction and block broadcast) – but it would keep the process execution logic and business rules off-chain, which may incur security/trust issues, as they could then be tampered with. Alternatively, a smart contract on the blockchain could be represented as another pool, which would be the orchestrator. In that case, all messages from participants would be relayed by the smart contract, possibly with computation by it (such as script tasks). However, that notation would be clumsy and more of a pure implementation artefact rather than a business process model that serves as a notational bridge between business experts and developers.

Choreography diagrams are another alternative to capture a process collaboration [51]. A choreography diagram captures how the participants in a collaborative process interact, i.e. the possible sequences of interactions. Choreography diagrams are intended to capture situations where "there is no central controller, responsible entity, or observer" [58, p. 23], but in our setting, the blockchain and the smart contracts it hosts conceptually play the role of a coordination mechanism. The focus of choreography is not the orchestration of a process, but to capture the behaviour of the participants based on the messages they sent and receive. In blockchain environments, participants share a common infrastructure, i.e., the

[2]https://www.bizagi.com/en

[3]https://camunda.com/

blockchain, which acts as an orchestrator. Indeed, the blockchain specificity hides the organizational separation modelled as BPMN pools in the standard, typically implemented through different systems.

A drawback of the blockchain platforms comes from their isolation insofar as smart contracts cannot invoke operations executed off-chain [172]. The latter hinders the notion of message exchanges when implementing collaborative processes. For example, a participant sending a message to another means that they write a transaction in the blockchain. The receiver can read the message, but also the other participants with access to a node in the network, i.e., everyone in public blockchains. Therefore, in practice, interactions between participants in blockchain environments do not necessarily materialize as message exchanges, but rather as transactions executed on the blockchain.

Aligned with the above, we decided to use process diagrams with a single pool for CATERPILLAR: it can represent the business process, and it makes use of blockchain for data storage, computation, and communication. This decision does not contradict the standard definition of collaboration and choreography diagrams. Over time, both single-pool models and collaborative/choreography models have been used to capture collaborative processes. For example, two of the earlier approaches to collaborative process modelling and execution were the Mentor and the Self-Serv systems, which relied on single-pool models, represented as statecharts (Mentor in the context of independent organizational units in an enterprise, and Self-Serv in an inter-organizational setting) [15, 166]. Indeed, choreographies can be translated into executable processes shared by all the parties [118, 161]. Also, in our approach, each organization still maintains its internal processes running off-chain, sharing on-chain only the parts of the process relevant to the collaboration as for choreography diagrams. The critical difference in our proposal is that, as the participants must write transactions in the blockchain, we replace the message exchanges by the execution of tasks, which in our opinion, are more convenient on the blockchain settings.

Due to one primary goal of CATERPILLAR is to automate the execution of business processes, it fits better to model the process as a set of flow elements instead of a collaboration or choreography diagrams whose goal is to represent the interactions between processes. Also, by representing the collaboration as a single process, we can include constructions like sub-processes and event propagation, which increases the process reusability and offer a broader range of options to the participants on the execution. Accordingly, the choice of single-pool models to capture blockchain-based collaborative processes was purely a matter of convenience, and it was not intended to exclude the use of choreography models (where the concept of interaction could potentially also be mapped to a transaction). In other words, we acknowledge that the modelling could start from a choreography model that can then be mapped into a single-pool model for execution.

In line with the above, the design of CATERPILLAR is driven by the following principles:

1. The collaborative process is modelled in the same way as an intra-organizational business process executed on top of a traditional BPMS. In other words, the collaborative process is modelled as if all the parties shared the same process execution infrastructure (the blockchain). Accordingly, the starting point for implementing a collaborative business process is a single-pool BPMN process model (not a collaborative process or a choreography in which parties communicate via messages). Each independent party in the process is represented as a lane. Hand-offs between parties are represented via sequence flows that go from one lane to another (and not via messages).
2. A collaborative process model may comprise sub-processes. Accordingly, an instance of a process may be linked to instances of sub-processes and vice-versa.
3. The full state of the process instance is recorded on the blockchain, and all the metadata required to retrieve the links between a given process instance and its related sub-process instances is also recorded in the blockchain.
4. All the execution logic captured in the process model is translated into smart contract functions, which can run independently of any other runtime component. In other words, the execution of process instances can proceed even if no instance of the off-chain runtime component is running. Also, several instances of the runtime component can be running at a given point in time (e.g., one instance per participant).

In our opinion, by representing the actors of a process as pools or lanes in the model, we do not capture the fact that these actors may be appointed or replaced at runtime. Also, in some cases, the participants need to update the control-flow of a process for a given instance at runtime, e.g., to handle exceptions or including paths not known at the design time (cf. see Section 1.1.3). Accordingly, apart from the the design principles outlined above CATERPILLAR considers a *flexibility* requirement. Specifically, we focus on two significant cases: (i) the system would allow dynamic binding and unbinding of actors to roles, and (ii) the parties may change the default routing logic of the business process at runtime.

Another primary requirement considered in CATERPILLAR comes from the space limitations existing on the blockchain platforms. This so-called *efficiency* requirement aims to reduce the size and reusability of the smart contracts produced from the process models (cf. see Section 1.1.2). Accordingly, the design of the system demands a modular architecture such that only the minimal set of components required for a full-fledged execution (cf. design principle 3) is handled on-chain. Similarly, the implementation of a process on-chain should take into consideration (when possible) compact data structures to cope with the storage limitations inherent to blockchain platforms. Besides, the computational complexity of the functions in the smart contracts implementing the process models may be constant or linear in the worst cases [39].

To recap, the goal of CATERPILLAR is to enable a set of parties to develop, deploy, and execute process-centric decentralized applications, or process-centric dapps, on a blockchain platform in a way that ensures compliance by design, meaning that no party is able to execute a transaction that does not abide with the collaborative process model. Indeed, CATERPILLAR also seeks to make the development and deployment of the process-centric dapp as seamless as possible by starting from a high-level specification of the collaborative process and automating the compilation and deployment of this specification into the blockchain.

1.3. Overview of the Contributions and Outline of the Thesis

Regarding the steps of the business process life-cycle [45], CATERPILLAR spans the implementation, execution, and monitoring, with support for adaptation and evolution of the process at runtime. Accordingly, and aligned with the research questions, design principles and requirements outlined in Section 1.2, the thesis makes three contributions as described below.

The first contribution addresses research question **RQ1**. We propose a compilation-based engine implemented by the CATERPILLAR system that translates hierarchical BPMN process models enhanced with data and resource constraints into smart contracts written in the Solidity language. Specifically, our compiler allows an extensive array of BPMN constructs, including sub-processes, multi-instance activities, event handlers and specialized activities. Additionally, our engine supports the deployment of smart contracts in the Ethereum blockchain, as well as the execution and monitoring of collaborative processes through a set of on-chain and off-chain components designed in a system architecture that requires no trust among the participants. Finally, an evaluation of the compilation-based engine demonstrates the trade-off between efficiency (consumption of the cryptocurrency Ether) and the ability to run inter-linked business processes entirely on the blockchain (without requiring external runtime components).

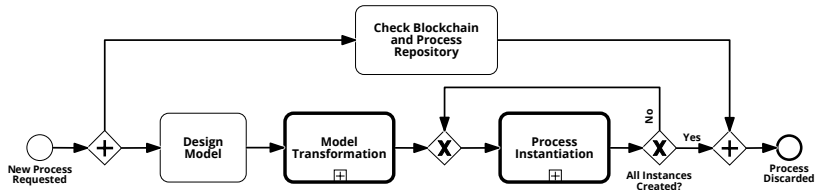
The second contribution focus on research question **RQ2**. We put forward a new approach to blockchain-based business process execution based on an interpreter of BPMN process models. Unlike compiled approaches, the interpreter encodes the semantics of the BPMN language in a singleton smart contract. As such, the interpreter needs to be deployed only once on the blockchain. Besides, the interpreter may be attached to multiple process models, each of which is represented using dynamically updatable and space-optimized data structures. Like the compilation-based engine, the interpreter supports the instantiation of any of its associated process models and allows participants to monitor the state of process instances and to execute tasks thereof. A modular architectural design, combined with the use of dynamic data structures, provides flexibility for the participants in the process to react to unexpected situations during its execution. Specifically, the interpretation-based engine allows participants to maintain different variants

of the same process model or to permanently modify a process model so that all running and future process instances follow the new version of the model. The proposed interpreter has been implemented as an extension of the CATERPILLAR system that comprises two process execution engines: a compilation-based and an interpretation-based engine. An experimental evaluation assesses the costs of the interpreted execution approach compared to existing compiled approaches.

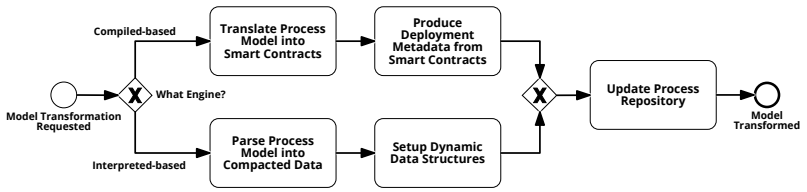
The rationale behind the first two contributions above deserves some further explanation. As discussed in Section 1.1.2, there is a trade-off between whether to use compiled or interpreted approaches on the process execution. On the one hand, compiled approaches exploit the immutability of the smart contracts, providing a more secure execution as participants cannot tamper the agreement made when designing the process model. This approach may be costly under scenarios in which flexibility is a requirement due to the need to redeploy the contracts after updating a model. On the other hand, interpreted approaches support a more flexible execution and are less costly as they reduce the size of the code and increases the reusability of the contracts. However, as the participants may update the process on their behalf, it may lead to trust issues or deadlocks if the updates are not performed correctly. Accordingly, the CATERPILLAR system provides two engines, i.e., a compilation-based and an interpretation-based engine, so the participants can decide based on their requirements which approach fits better based on the process and the blockchain in which the contracts will be deployed.

The third contribution responds to research question **RQ3** and addresses the lack of flexible control mechanisms in existing blockchain-based approaches for collaborative business process execution. We propose three types of controlled flexibility mechanisms: (i) dynamic binding of actors to roles in a collaborative process; (ii) dynamic selection of sub-processes; and (iii) dynamic selection of alternative pathways in a given execution state of a process. 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. We also propose an approach to analyse policy specifications for dynamic role-binding in order to prevent circular dependencies that may prevent one or more roles from being bound to an actor under some circumstances. Finally, we show 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. Besides, we implement the proposal in the CATERPILLAR system and show the cost to deploy and execute the smart contracts on the Ethereum blockchain via an empirical evaluation.

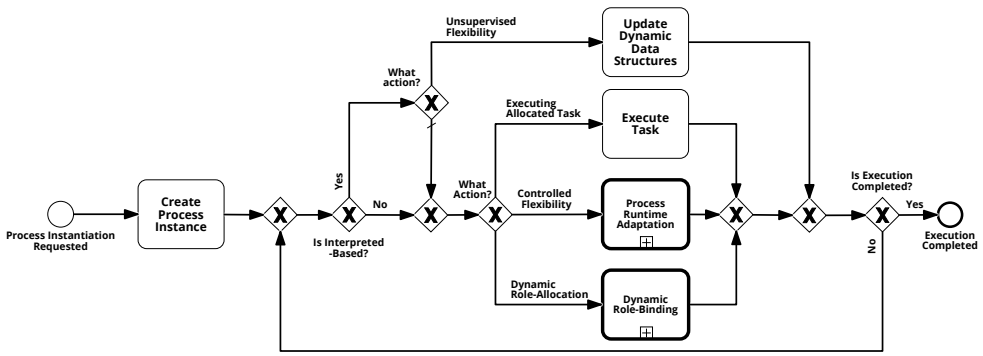
The BPMN model in Figure 2, provides an overview of how the contributions of this thesis are integrated into the CATERPILLAR system. First, any of the participants of the collaborative process design the joint BPMN model, serving as an agreement among the participants (as shown Figure 2 (1)). Then, also any of the



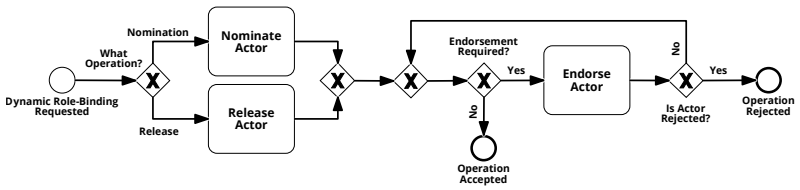
(1) Overview of a Process Execution on CATERPILLAR



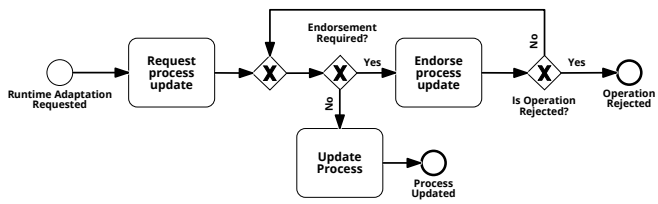
(2) Sub-process: Model Transformation, from BPMN models to smart contracts



(3) Sub-process: Process Instantiation, creation and execution of process instances



(4) Sub-process: Dynamic Role-Binding, allocating actors to roles at runtime



(5) Sub-process: Process Runtime Adaptation, controlled flexibility on updating the control-flow

Figure 2: Overview of the process execution on CATERPILLAR

participants use CATERPILLAR to transform the corresponding model into smart contracts which later will enforce the execution of the process instances. On the transformation of the process, Figure 2 (1), the participant can choose either the compilation-based or interpretation-based engine for the process execution (see Chapters 4 and 5 for further details). Then, CATERPILLAR stores all the metadata regarding the process transformation in a decentralized process repository (off-chain) which is shared by all the participants. Note that, each participant can check the information from the process repository and the blockchain at any time. Besides, once transformed, the generated artefacts can be used on the creation of as many process instances as required (Figure 2 (1)).

Once the process model is transformed, any of the participants may create one or many instances of the process (Figure 2 (3)). To that end, the participant, named case creator, may use the CATERPILLAR's user interfaces or may create the instances directly through the blockchain, i.e., using the metadata produced either from the compilation or the parsing of the model. Accordingly, the blockchain address produced when the transaction is appended will serve as the identifier of the corresponding process instance. Then, during the execution of a process instance, the participants have different options: bind actors to roles dynamically, update the control-flow of the current process instance, or to execute their tasks which are allocated as work-items.

As discussed in Section 1.1.2, compilation-based solutions face flexibility issues compared to interpretation-based ones. Notwithstanding this, CATERPILLAR supports some flexibility for compilation-based approaches, which is achieved through late-binding and late modelling of processes [141] and the factory pattern [54], i.e. flexibility by looseness [128]. For example, both engines of CATERPILLAR relates call-activities to instances of smart contracts implementing the corresponding sub-processes, which can be linked at any time during the execution. Accordingly, in this thesis, we propose agreement policies to handle flexibility by looseness mechanisms, which works for both the interpretation-based and the compilation-based engines, and which do not introduce deadlocks and therefore do not require additional verification techniques to be put into place. These mechanisms, referred to as controlled flexibility in Figure 2 (3) are further discussed in Chapter 6.

Other flexibility mechanisms, such as adding, skipping or removing elements in the process model, are possible only on the interpretation-based execution. For example, according to the taxonomy presented by Reichert and Weber [128], there are four major control-flow based flexibility requirements: adaptation, evolution, variability and looseness. In that regard, the interpretation-based engine supports those four requirements. However, the variability, adaptation and evolution requirements, labelled as unsupervised flexibility in Figure 2 (3), may lead to deadlocks on the interpretation-based engine. Thus, such updates on the dynamic data structure of the interpreter can be performed for a given process instance at runtime if and only if all the participants agree on it as they entail a higher risk to

deviate the process execution. More specifically, process participants are responsible for verifying off-chain whether those unsupervised changes are consistent before propagating them to the blockchain. To that end, participants can either perform the consistency validation automatically (by using private-owned tools) or manually.

Figures 2 (4)-(5) outlines how participants can enforce the flexibility mechanisms by consensus (see Chapter 6). First, participants can nominate or release actors into roles during the process execution, subject to the endorsement of other participants. For example, after creating a process instance, the case creator can appoint and actor to play a role in that instance, and accordingly execute the tasks granted to that role. The dynamic role-binding schema is described by polices, which restricts how the actors can be nominated or released, and who must endorse that binding. Similarly, agreement policies restrict the control-flow flexibility mechanisms (Figure 2 (5)). Accordingly, the participants can decide by consensus whether to update a process at runtime, e.g., to enforce the late-binding of a sub-process. In other words, CATERPILLAR promotes an approach of flexibility by underspecification [141] or looseness [128].

The contributions of this thesis have been previously documented in publications I-V, referenced at the end of the thesis (see "List of original publications"). In addition, the code of the CATERPILLAR system is open-source and can be accessed from a public *GitHub* repository (see Appendix A).

The rest of this thesis is structured as follows. Chapter 2 introduces the relevant concepts and principles from business process management and blockchain. Chapter 3 reviews the state of the art of existing methods and tools related to this thesis. Chapter 4 provides a detailed description of the architecture design, implementation and evaluation of CATERPILLAR's compilation-based engine. Chapter 5 proposes and evaluates CATERPILLAR's interpretation-based engine, also including the functioning of dynamic data structures and the space-optimized representation of business processes. Chapter 6 describes and assesses the controlled flexibility mechanisms for blockchain-based execution of collaborative business processes. Finally, Chapter 7 concludes the thesis, summarizing the core contributions and providing future work directions.

2. BACKGROUND

In this chapter, we outline some relevant concepts of the research domain covered by this thesis. First, Section 2.1 introduces the fundamentals of Business Process Management (BPM) described by the process life-cycle. Next, Section 2.2 dives into the representation of process models written following the BPMN standard that is supported by CATERPILLAR. Finally, Section 2.3 introduces key elements like consensus protocols and smart contracts to illustrate blockchain functioning and characteristics. We emphasize the Ethereum blockchain as the platform that supports the current version of CATERPILLAR.

2.1. Business Process Management

Business Process Management (BPM) is a discipline encompassing the principles, methods, and tools which allow organizations to analyze, execute, and monitor business processes [45]. Specifically, BPM guides how stakeholders interact and collaborate to fulfil organizational goals across the entire process life-cycle: identification, discovery, analysis, redesign, implementation, execution, monitoring, and adaptation. At the identification stage, organizations extract high-level descriptions and relations that are relevant to a problem within a given scope from a process-oriented perspective. Next, in the discovery phase, analysts collect information about how the actual process operates into a graphical representation, which is called the as-is model. Then the analysis phase aims to identify issues in the process and opportunities for improvement. As a result, in the redesign stage, the as-is model is translated into a to-be model, which eventually is ready to be implemented and executed.

After redesigning the process, the implementation phase deals with transforming the to-be model into software components to execute the process. The last may entail configuration/implementation tasks required to automate the process as an information system and changes in the way that participants interact with it. Process execution includes the creation and handling of individual process instances, usually referred to as process cases. Often, business processes are implemented by BPMSs, which also facilitate process monitoring, i.e., collecting and displaying events during the execution and notifying participants about undesired behaviours. In CATERPILLAR, we focus on these three phases: implementation, execution, and monitoring. We also support the adaptation and evolution of the process at runtime; i.e., the participants can change a process during its execution.

2.2. Business Process Model and Notation

Process models are an essential element of BPM. They provide high-level abstractions that simplify the understanding of the process among the participants involved. Additionally, the standardization of processes as to-be models is crucial

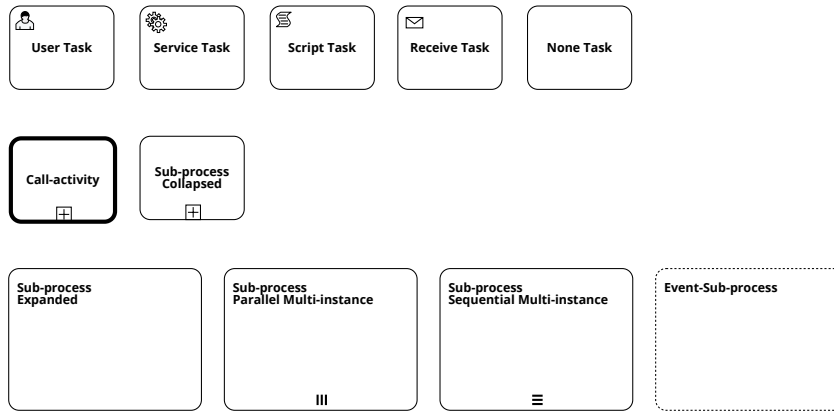


Figure 3: Example of the activities supported by CATERPILLAR.

to automate their execution by process-aware information systems. In the this section, we introduce the process modelling language that is most widely accepted by practitioners in the industry, which we also adopted in CATERPILLAR: Business Process Model and Notation (BPMN) [58].

BPMN is a graphical standardization of business processes that creates a bridge between process design and implementation. It includes activities, events, gateways, and arcs, which are called sequence flows. BPMS also supports data objects and resources, e.g., represented as pools/lanes. However, in this section, we will refer only to the control-flow related elements, as in CATERPILLAR, we use other representations for data and resources that, in our opinion, are more suitable in a blockchain-based execution. The remaining of this section describes some of the core control-flow elements handled by CATERPILLAR. For further details, we refer readers to the BPMN 2.0 specifications [58].

Activities represent work to be performed within a process by a resource, which can be either human or an information system. Activities can be atomic, referred to as tasks, or non-atomic, which refers to sub-processes and call-activities. Non-atomic activities allow the inclusion of other activities. Thus they enclose the set of graphical elements that conform to a process hierarchy. Figure 3 illustrates the types of activities we handle in CATERPILLAR[1]:

– TASKS are atomic activities that, based on their inherent behaviour, can be classified in different types. (i) User Tasks are typical units of work performed by humans as work-items managed by a workflow system. (ii) Service tasks are executed by services, e.g., web services or automated applications. In CATERPILLAR, those services are running off-chain. Thus the interaction is managed via oracles [64]. (iii) Script tasks are performed internally by the workflow system; i.e., they execute on-chain scripts em-

[1]Modelling an activity also follows naming conventions, e.g., the label of a task represents its goal in the form of an imperative verb followed by a noun. Figure 3 does not obey the naming conventions as its purpose is illustrating the types of activities.

bedded in smart contracts on blockchain-based applications. (iv) Receive tasks are designed to wait for messages arriving from an external participant. In our solution, as we avoid message exchanges among participants, receive tasks offer a syntax sugar which allows participants to select an outgoing path in event-based gateways (see the receive task outgoing from the event-based gateway in Figure 5). (v) None or Default tasks are mainly used with design purposes, e.g., in as-is models. As they have no execution semantics, when they are used in to-be models, they should be transformed during the implementation stage. Otherwise, they will be skipped when reached in the control-flow.

- SUB-PROCESSES are activities containing other activities, events, gateways, and flow arcs that conform to a process embedded as part of a bigger one. Sub-processes allow us to organize a process as a hierarchy and to restrict the scope of the process variables and events. Sub-processes can either be expanded, i.e., if all steps of the process are displayed, or collapsed to show a high-level overview of the process, i.e., only the name and a plus-sign are displayed.
- CALL-ACTIVITIES serve as a wrapper for invoking a process that is external to the current one. Precisely, call-activity points to a sub-process; thus, when the execution arrives at the call-activity, it creates a new instance of the linked sub-process which runs in parallel as a child of the current process. Then the parent process instance waits to continue its execution until the sub-process ends. Call-activities are an essential tool as they allow the reusability of processes. Note that, conceptually, the only difference between sub-process and call-activity is that the former one is embedded and the other refers to an external definition.
- EVENT-SUB-PROCESSES are specialized sub-processes that are triggered by a start event. An event-sub-process has no incoming/outgoing sequence flows. It is scoped into a process/sub-process in the hierarchy. Thus it can be triggered at any time within an active instance of its parent process/sub-process. There are two types of event-sub-processes. An interrupting event-sub-process stops the execution of the process/sub-process where it is scoped. In contrast, a non-interrupting event-sub-process runs in parallel to its parent sub-processes. Event-sub-processes are often used to handle exceptions. For example, the occurrence of a problem during the execution of a process can propagate an error event. Then, this error can trigger an event-sub-process to cancel the execution in the current scope and to react accordingly to the problem.
- MULTI-INSTANCE ACTIVITIES allow repeating a given task/sub-process sequentially or in parallel during the process execution. Parallel multi-instance activities contain three short vertical lines in the bottom, while three horizontal lines represent sequential multi-instances. When the exe-

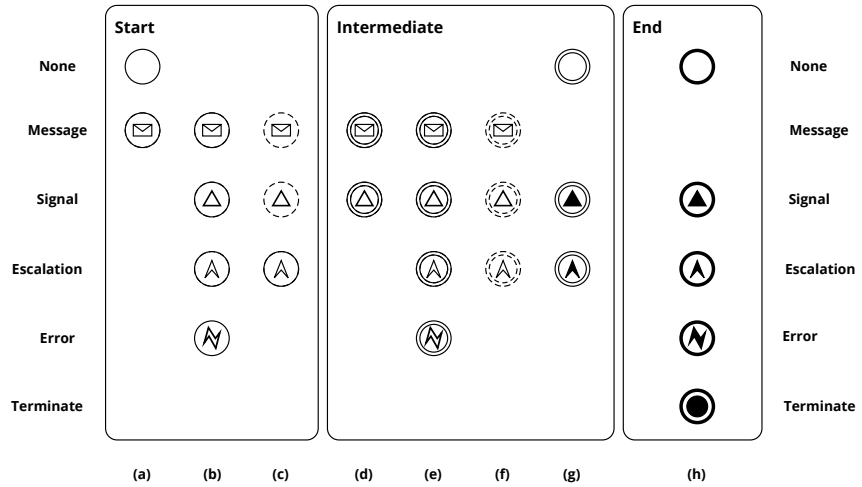


Figure 4: Events supported by CATERPILLAR: (a) Start top-level process, (b) Start event-sub-process interrupting, (c) Start event-sub-process non-interrupting, (d) Intermediate catching, (e) Intermediate boundary interrupting, (f) Intermediate boundary non-interrupting, (g) Intermediate throwing, (h) End.

cution arrives at a parallel multi-instance activity, it creates a certain number of concurrent instances of the activity. Sequential multi-instance activities function as a loop where the next instance of the activity is created after the completion of the previous one. A multi-instance activity ends after the completion of all its instances.

Events, which are modelled as circles in BPMN, describe something that happens during the execution of a process. They may change the execution flow of a process, and they often have a cause and an impact. There are three types of events: (i) START EVENTS, which trigger the instantiation of a (sub-)process, (ii) INTERMEDIATE EVENTS, which occur during the process execution, and (iii) END EVENTS, which end a path or an entire (sub-)process. Within these three types, events can throw a result or catch a trigger. Those so-called triggers are represented by a symbol that identifies the cause/effect of the event, e.g., message, escalation, signal, error, etc. (see Figure 4).

Start events always catch a trigger, and they have no incoming sequence flows. Indeed, each (sub)-process in the hierarchy starts when its corresponding start event is triggered. There are some differences regarding whether the events start a top-process, a sub-process, or an event-sub-process. The standard allows top-level processes to have multiple starting events; however, in CATERPILLAR, we consider only processes with a single start event. In addition, top-level processes and sub-processes are always started by a none event, i.e., with no trigger defined. On the contrary, event-sub-processes allow several triggers (see Figure 4). Also, the start events can be *interrupting*, i.e., triggering the event terminates the instance of the (sub-)process containing the event-sub-process, or *non-interrupting*, which

instantiates the event-sub-process in parallel with its parent.

We adopt the idea of tokens traversing sequence flows to illustrate the process execution. After triggering a start event, it generates a token on its outgoing sequence flows. Then the execution proceeds such that an element can be executed (is enabled) if its incoming sequence flows contain the required tokens; e.g., an activity is enabled if its incoming sequence flow contains a token. Executing an activity consumes/produces tokens in its incoming/outgoing sequence flows. Finally, a process is completed if no token exists in any of the sequence flows.

Intermediate events can be either throwing or catching. They can be placed either as part of a flow path, i.e., containing incoming and outgoing sequence flows, or as a boundary of an activity, having only an outgoing sequence flow. Intermediate events affect the execution flow, e.g., changing the normal flow to handle exceptions, but they do not directly start/end the process execution. Also, boundary events have no incoming sequence flows and can only catch triggers, and as a result, they can interrupt or not the activities to which they are attached. Like the activities, intermediate events consume/produce tokens on their incoming/outgoing edges when they are thrown/caught.

End events always throw a result, and they have no outgoing sequence flows. An end event always consumes the token from its incoming sequence flow, but it never generates new ones. Commonly a process has multiple end events. Thus, the throwing of an end event does not necessarily terminate a process instance, unless the event consumes the last token. However, the triggers of some end events explicitly finish the process, i.e., removing the remaining tokens.

The strategy to forward an event depends on the type of trigger. (i) MESSAGES, as the name suggest, are used to send (throw) or receive (catch) messages to/from process participants. [2] (ii) ERRORS represent an anomaly during the execution. After throwing an error event, it propagates to the ancestors of the origin sub-process until either catching the event or reaching the top-level process. In case the event is caught on the boundary of an activity (sub-process), it terminates the corresponding activity instance. If the error triggers the starting of an event-sub-process, it terminates the current instance of the (sub-)process containing the event-sub-process. Finally, if the trigger arrives at the top-level process (without being caught), it terminates the entire process. (iii) ESCALATIONS, also propagates the trigger to the parent sub-processes. This trigger works like error events, with the difference that catching an escalation not necessarily terminates the corresponding sub-process, i.e., the catching event can be either interrupting or non-interrupting. (iv) SIGNALS broadcast the trigger to the entire process hierarchy, i.e., not just to the ancestors of the origin sub-process. Like the escalation, catching a signal may or may not interrupt the enclosed sub-process. (v) NONE

[2] CATERPILLAR avoids the exchange of messages between participants. However, the system allows attaching message events to activities (to interrupt them), or as an output of event-based gateways (to allow participants to choose a path). For completeness, this section introduces all the messages events described in the BPMN standard.

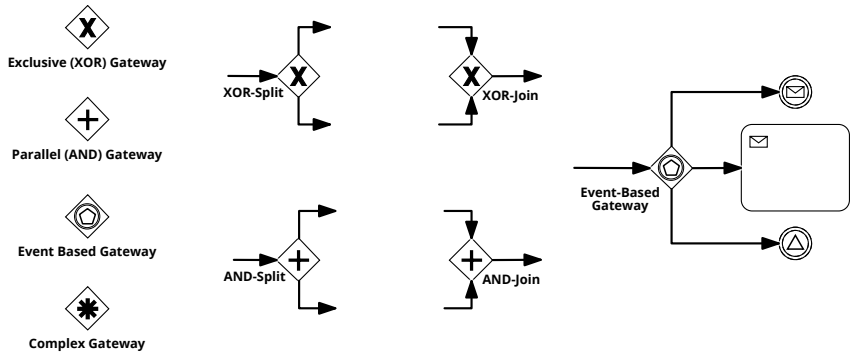


Figure 5: Example of the gateways and flow as supported by CATERPILLAR.

events have no trigger. They mainly serve either to start (sub-)processes or to notify a parent sub-process about the termination of a child. They cannot be used to trigger event-sub-processes or for catching intermediate events. However, they can be throwing intermediate throwing events in the normal flow, e.g., to indicate some change in the state of the process. (vi) TERMINATE can be used as end events only. They force the ending of all the activities, thus terminating the entire (sub-)process. See Figure 4 for the full list of events that CATERPILLAR supports.

Gateways control how sequence flows converge and diverge in a process, e.g., to model decisions based on data and events. Regarding the number of incoming and outgoing sequence flows, gateways can be divided into splits and joins. Split gateways have one incoming and multiple outgoing sequence flows. In contrast, join gateways have multiple incoming and one outgoing sequence flow. The type of a gateway determines when tokens arrive, their consumption and the generation of new ones in the outgoing sequence flows. Figure 5 illustrates the types of gateways supported by CATERPILLAR, which are represented by a symbol within a diamond shape, and how they process the tokens during the process execution.

- EXCLUSIVE (XOR) GATEWAYS model alternative paths within the process flow. The XOR-split expresses that for a given process instance, only one path (one outgoing sequence flow) can be activated. The decision relies on data conditions, which are typically modelled by a question as to the label of the gateway and the possible answers as condition expressions on the outgoing sequence flows. Thus, when a token arrives at the XOR-split, the execution moves to the outgoing sequence flow that fulfils the condition expression based on the process data. Indeed, XOR-joins merge alternative paths. Thus, XOR-join gateways consume an incoming token, generating a new token in the outgoing sequence flow, without synchronization; i.e., only one incoming token is required to activate the XOR-join gateway.
- PARALLEL (AND) GATEWAYS create and synchronize parallel sequence flows, i.e., they model concurrent behaviour. When a token arrives at an AND-split gateway, the execution continues in parallel through all the outgoing sequence flows. The AND-join synchronizes parallel paths; thus,

they are activated when all incoming sequence flows have a token.

– EVENT-BASED GATEWAYS model alternative paths which are activated by an event. Specifically, when one a token arrives at the gateway, it remains active until one event is caught, moving the execution to the corresponding outgoing sequence flow. Unlike XOR-splits, in which the decision relies on data, event-based gateways rely on actions triggered by events, e.g., a message received by a process participant.
– COMPLEX GATEWAYS, as the name suggests, model complex synchronizations not captured by the other gateways in the process flow. This type of gateway allows rules to specify how the tokens are consumed/produced. For example, it is possible to determine at runtime, which incoming sequence flows activate the gateway or how to proceed with the execution by following one or many sequence flows.

2.3. Blockchain Technology

A blockchain is an immutable append-only ledger replicated across a network of untrusted peer nodes. The ledger is represented as a linked sequence of blocks which contains an ordered set of transactions, as shown in Figure 6. Each block is chained to the previous one by its hash value. Thus, the only way to alter/delete a transaction is by reconstructing the entire chain. Some nodes, called miners, are responsible for validating and grouping transactions submitted by the users into blocks appended to the blockchain. As no central authority exists, the miners must reach consensus in a distributed manner [173].

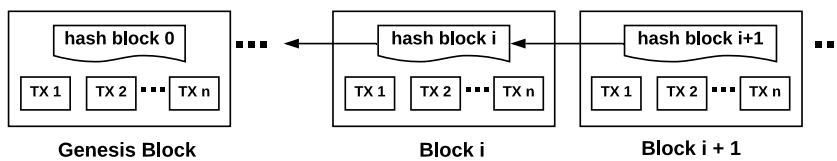


Figure 6: Blockchain: Blocks and Transactions (adapted from [181]).

Clients use a blockchain system (a concrete network, as shown in Figure 7) 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 adequately formed and signed by its creator. No trust in individual clients or nodes is required, as the transactions are cryptographically signed and validated and are broadcast widely. A consensus mechanism ensures that the transactions are tamper-proof without assuming mutual trust between participants [40].

2.3.1. Types of Blockchains and Consensus Protocols

Existing blockchains are typically included in one of the following three categories. (i) Public blockchain networks allow open access to anyone in the world.

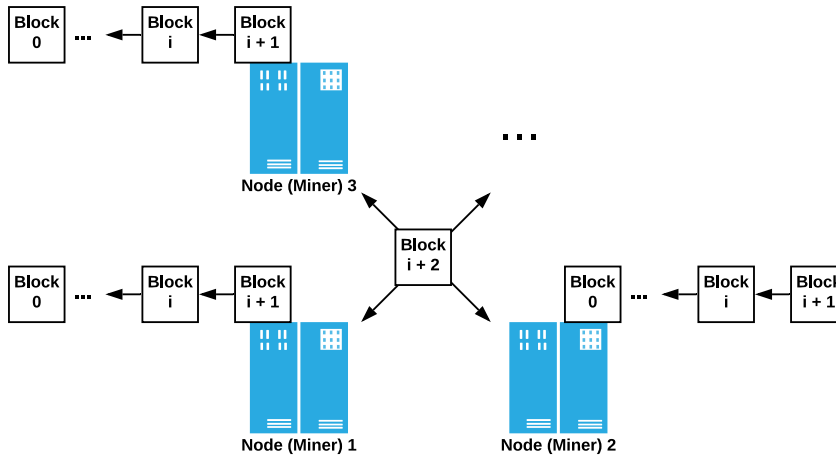


Figure 7: Untrusted peer-to-peer network representing the blockchain.

In other words, anyone can submit and access the transactions and participate in the consensus protocols. (ii) Consortium blockchains restrict the consensus protocols to a pre-selected set of nodes, i.e., across multiple organizations. However, submission of and access to the transactions can be either public or limited to a set of participants. (iii) Private blockchains are governed by a single organization that decides about permissions of the participants to submit/read transactions [174].

Public blockchains are also known as "permissionless" because they are open and decentralized. In addition, there is no central party that manages the membership or which can remove members from the network. 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 very 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 [168, 181]. Bitcoin [107] and Ethereum [23] are two of the most prominent examples of public blockchains.

Consortium and private blockchains are also called "permissioned" because a central authority decides on the rights of the participants in the network. These blockchains are more efficient than the permissionless ones as they require fewer validators. Similarly, permissioned blockchains offer more privacy and reduce redundancy as they have less data replication. However, these blockchains may be susceptible to trust issues, given that they are partially centralized. For example, the network can be tampered with if the majority of the organizations in a consortium agree to it [168, 181]. Hyperledger Fabric [27] and R3 Corda [19] are among the most relevant permissioned blockchains.

Most public blockchains, e.g., Bitcoin and Ethereum, use proof-of-work [55] as a consensus mechanism, whereby the creation of a new block, which is referred to as mining, requires solving a computationally difficult cryptographic puzzle. A

node uses its computational power for mining blocks. Mining the next valid block requires solving the puzzle before any other miner and is rewarded financially. Each block contains the hash value of the previous block, thus linking the blocks in the database. Moreover, any attempt to alter a block would incur high computational costs: preserving the links established by the cryptographic hashes would require the recreation of the whole set of subsequent blocks from the altered block. The economic factor of block rewards and the resulting computational difficulty in altering a block makes tampering virtually impossible.

The main drawback of proof-of-work is that it incurs a very high consumption of resources and energy due to a lot of calculations being performed by the miners. Accordingly, other energy-saving protocols have been emerging. For example, Ethereum aims to overlay proof-of-work with a new proof-of-stake system named Casper [24]. Proof-of-stake algorithms [74] rely on the amount of cryptocurrency the miners own instead of investing in computational resources. Mainly, miners run a process which randomly chooses one miner proportionally to the amount of stake they own. The protocol promotes the idea that participants with more currency have less probability of attacking the network. However, this strategy may lead to the wealthiest miner dominating the system, and it can be more vulnerable to attacks because the mining cost is almost zero [181].

2.3.2. Ethereum Blockchain and Smart Contracts

Smart contracts are one of the novelties that the Ethereum blockchain introduced. A smart contract is a computer program deployed on the blockchain which may be invoked via a transaction [173]. When a new block is mined and broadcast, each full node in the peer-to-peer network is required to locally execute the transactions, including the ones executing the smart contracts, and perform the respective calculations to derive the state after the execution of the transactions included in that block. Applications that are designed to provide their main functionality through smart contracts are called dapps [173].

Smart contracts are executed over the Ethereum Virtual Machine (EVM), which is bundled within each peer node. The EVM is a runtime component which provides a stack-based computing platform with a small set of operations, and that is sufficient to support the definition of Turing-complete programming language. The size of the EVM word is 256-bits. For each contract, the EVM allocates a persistent memory, referred to as storage, which is organized as a key-value store that maps 256-bit words to 256-bit words. The persistent memory is private, and it cannot be directly accessed by another contract or transaction. Moreover, a contract gets access to volatile memory with each function call, which can be expanded by one word at a time, and that serves to store intermediate values. The EVM uses a stack and no registers to execute the instructions of the contracts. The stack has a limit of 1024 words, and only the topmost 16 words are accessible at a given moment in the execution; hence the need for volatile memory. Finally, each contract

can write data into a log which is visible to external applications.

Several contract-oriented programming languages and compilers thereof have been developed that produce bytecode for the EVM. Among them, we have selected Solidity for our development, because it is the most widely used and supported. Solidity is a strongly typed language, and its syntax resembles JavaScript. A contract in Solidity is defined in a similar way as classes in Java-like object-oriented languages. Thus, the definition of contracts usually includes persistent properties (i.e., the contract's state) and functions to query and manipulate them.

Each peer in the network communicates with other peers using the Ethereum wire protocol [167]. The details of this protocol are outside of the scope of this thesis. In addition to the internal interactions, each peer exposes a number of methods over an RPC-style endpoint, which is known as the Ethereum's JSON-RPC API,[3] because it uses data exchanges formatted according to JSON-RPC specifications.[4] It is this RPC endpoint that external applications (e.g. wallets and other software) use for interacting with the Ethereum blockchain.

From a technical point of view, a (smart) contract corresponds to the code that is deployed to the Ethereum blockchain. The cost to deploy a contract in Ethereum, which is proportional to its bytecode size, is measured in a unit called *gas*. Once deployed (or instantiated), the smart contract is related to a unique hash address that can be used by external applications to invoke the public functions of the smart contract. Such invocations generate transactions whose cost (also measured in gas) depends on the number and type of the executed instructions [167].

Generally speaking, the interaction between an external application and a contract can happen in two ways:

1. An external application requests the execution of a transaction by calling on a contract's function. The transaction is then forwarded to the network of peers. A transaction is seen as executed only if a block includes the transaction in the chain. Since this type of interaction requires block mining, on public blockchains, the requester typically pays a transaction fee.
2. An external application can request the execution of a contract's function on a single Ethereum node, that is, without forwarding a transaction to the network of peers. Since no block mining is required, this type of interaction incurs no fee. This type of interaction can be used for querying the current contract's working memory state or for previewing the outcome of executing a contract's function given the current state. Moreover, external applications cannot access the contract's working memory state unless the contract provides public functions permitting it.

In contrast, smart contracts cannot to call external programs. However, as mentioned before, a contract can write information in a log that is visible to external applications. Moreover, the JSON-RPC API provides some operations that can be

[3] https://github.com/ethereum/wiki/wiki/JSON-RPC

[4] https://www.jsonrpc.org/specification

used to install log-filters on a local peer that can be repeatedly polled to retrieve the entries added to the contract's log. This way of interaction is widely used to implement a sort of push-oriented interaction with external applications and to forward requests to so-called blockchain oracles [22, 64].

3. STATE OF THE ART

Blockchain is an emerging technology for software applications in cross-organizational settings. The current efforts on integrating blockchain into BPM are still nascent at this stage [99, 131, 156]. However, some early works are starting to exploit the potential of blockchain as a tamper-proof platform for business process execution. This chapter discusses the existing solutions in the domain of application of CATERPILLAR. First, Section 3.1 explores the traditional architectures of BPMSs and their limitations on blockchain environments. Next, Section 3.2 explores existing solutions for implementing and executing collaborative business processes on the blockchain. In CATERPILLAR, we integrate the modelling/visualization tools provided by Camunda;[1] thus, the solutions in those areas are outside of the scope of this thesis. Finally, Section 3.3 analyses existing solutions that address the problem of flexibility in collaborative processes.

3.1. Architectures of Business Process Management Systems

BPMSs have been traditionally built upon the architecture described in Figure 8. An execution engine (core component) creates and handles the process instances, providing functionalities to store/retrieve data required by the process execution automatically, and to distribute work among participants, i.e., work-items which are offered/committed to/by the participants via the worklist handler. Indeed, BPMSs also provide functionalities to create and modify process models, to manage and monitor operational matters, to interact with external services [45], etc.

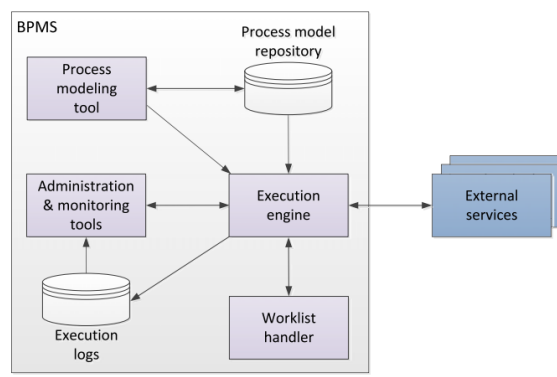


Figure 8: Basic architecture of BPMSs (from [45]).

The first reference architecture providing high-level principles and standards for BPMSs was presented by the Workflow Management Coalition (WfMC) in 1995 [66]. Since then, the WfMC model has been a well-established standard, although the arrival of new technologies has introduced updates to the original design [31, 165]. In the last decade, Service-Oriented Architectures (SOA)

[1] https://camunda.com/

emerged as a solution to develop more flexible and agile process-aware information systems [5, 14, 111, 134]. More recently, several cloud computing architectures [32, 117, 179] have been proposed, which reduce costs while maintaining high levels of efficiency [33]. The most recent reference architecture we found in the literature presented by Pourmirza et al. [124] is BPMS-RA. The authors rely on an extensive study [123] of existing commercial and academic BPMSs architectures to come up with a new proposal that spans more than 400 components, which was implemented by the GET service platform [13].

Classic approaches usually orchestrate business processes through a centralized BPMS, which poses several challenges in collaborative processes given organizational boundaries [18]. Recent solutions show the advantages of using distributed peer-to-peer architectures in inter-organizational processes. Among those, Martin et al. enact the control-flow of the Web Service Business Process Execution Language (WS-BPEL) in a decentralized manner [98]. Also using a distributed architecture in WS-BPEL, Li et al. demonstrate that an agent-based execution scales better than a non-distributed approach [85]. Other solutions consider distributed workflow-management systems to deal with the performance bottlenecks existing in traditional client/server architectures [151] and to improve the scalability and flexibility inherent in the strong coupling between process management and business application in centralized systems [61]. Similarly, Lei et al. propose a distributed artefact-centric BPM framework based on Representational State Transfer (REST) principles [84] to tackle scalability issues. These systems use the decentralization of components with optimization purposes, e.g., to improve parameters like scalability and throughput [172]. On the contrary, in blockchain platforms, throughput is a permanent issue due to the latency caused by mining. Note that blockchain-based systems use decentralization as a source of trust, which requires a different architectural design.

A recent survey [123] found that only 30% of the BPMSs analyzed support inter-organizational processes. The authors highlight the lack of trust between participants as one of the main challenges. At the same time, collaborative processes need to satisfy competing demands coming from their participants and hence need to incorporate some execution flexibility [57]. The latter is particularly the case in open and dynamic environments where the actors may switch from one process instance to another and may even change during the execution of one process instance. Unlike existing approaches, CATERPILLAR is a BPMS focused on the dynamism of collaborative processes with an architecture that combines the blockchain and external components to support flexible process execution in untrusting environments.

3.2. Blockchain-Based Collaborative Business Processes: Implementation and Execution

The first work addressing the problem of lack of trust in collaborative business processes using blockchain was reported by Weber et al. [163]. The authors present an approach for model-driven engineering (MDE) of collaborative business processes on top of the Ethereum blockchain platform. Specifically, the authors propose to compile a BPMN choreography diagram into Solidity contracts that ensures that the parties can only record their message exchanges in a way that is compatible with the ordering relations captured in the BPMN choreography diagram. The solution also introduces additional tasks for payments and data transformation, in extension of the BPMN standard.

Prybila et al. [125] present an alternative approach to monitor business processes executed on top of the Bitcoin blockchain via specialized tokens. Like in Weber et al. [163], the authors assume that the collaborative process is modelled as a choreography. In other words, both of these approaches assume that the parties in a collaborative business process interact via message exchanges, and they use the blockchain platform to record the message exchanges and to check or enforce that these exchanges occur in a specific order. This effectively means that the blockchain platform serves as one execution component of a collaborative process. However, many of the components are off-chain, and the interactions between parties occur via message exchanges.

Garcia et al. [56], propose an approach to transform BPMN process diagrams into Solidity smart contracts. This latter work does not assume that the parties communicate via message exchanges – but instead, the parties use the blockchain party as a coordination mechanism to maintain the state of the process and to determine what tasks or events may occur next, given the current state. The emphasis of this latter work is on optimizing the generated Solidity code in order to reduce the costs related to the deployment and execution of smart contracts. However, this work is restricted to flat BPMN process models consisting only of tasks and essential gateways (`AND` and `XOR` gateways). This previous approach does not support sub-processes, boundary events, or multi-instance activities.

Hull et al. [70] discuss a vision of how business process modelling and, more specifically, the Artefact-centric paradigm [38, 109] would be a suitable approach to model collaborative business processes executed on top of blockchain technology. Similarly, Norta et al. [110, 112] advocate the use of blockchain and smart contracts to coordinate collaborative business processes based on so-called choreography models, which are similar to collaboration diagrams in that they rely on the assumption that parties interact via message exchanges. Another related work [53] proposes mapping from a domain-specific language for "institutions" to Solidity. These previous research efforts only outline a possible architecture for modelling and executing blockchain-based business processes, but they do not provide any suggestions for implementation or evaluation. In contrast, Sturm et

al. [148] provide a model for implementation. However, their approach expresses control-flow solely via "requires" relationships, i.e., specifying which tasks are required to be completed before a given task can be executed. This allows expressing `AND` and `OR` gateways, but it does not disable alternative, un-executed tasks when executing an `OR` join.[2] By comparison, CATERPILLAR follows a different approach and supports a much wider range of BPMN elements, including subprocesses, multi-instance activities, boundary events, and several types of BPMN tasks.

Lorikeet [150] is a MDE tool that implements the approach of Weber et al. [163] and the translation algorithm from Garcia et al. [56]. The process components are supplemented with asset registries (e.g., tokens representing coins or titles). Lorikeet has been successfully applied in some projects with industry [150], demonstrating the value of process-oriented smart contract generation. The emphasis of Lorikeet is on the MDE approach, in which the generated code can be used as a basis for implementation; Lorikeet is not a BPMS.

Ladleif et al. extend the BPMN choreographies to provide a more suitable operational semantic relying on the blockchain capabilities [82]. The authors also provide critical analysis of the limitations of choreographies regarding the shared logic and on-chain data storage existing in blockchain platforms. This approach promoting message exchanges among participants (typical of choreographies) opposes our solution, which considers the blockchain as a shared execution infrastructure. Also their with focus on BPMN choreographies, Haarmann et al. generate smart contracts from Decision Model and Notation (DMN) models for implementing collaborative business decisions [59, 60]. Their proposal only considers the execution of decisions, putting aside other elements of the process execution.

Industry practitioners and BPMS vendors have also considered the idea of executing business processes using blockchain technology and smart contracts. For example, Rikken [130] discusses some perceived advantages of executing business processes via smart contracts. Bonitasoft [119] provides a software connector that enables process instances running on the Bonita BPMS to execute transactions on a blockchain. With this connector, a task in a BPMN process model can be configured in such a way that its execution generates a transaction on a blockchain. Similarly, [6] shows how to use IBM's BPM system to execute business processes on top of the Hyperledger blockchain platform. In this latter approach, data objects are kept in a (permissioned) blockchain, and tasks in the process read and write into these objects. This approach is suitable when one or a handful of tasks in a process needs to execute transactions involving untrusting parties or need to leave a tamper-proof trace of their execution. When permissioned blockchains are used (which are more scalable than public ones), it becomes practical to use this approach to record every execution event (e.g.,

[2] `https://github.com/Jonasmpi/PExSCo/blob/master/contracts/ContractCollaborationManager.sol`, commit a26274c9d564f8a2cc2477eaa26f633b485323ca, lines 107-122; accessed 2020-02-26

events indicating the start and end of each task) on a blockchain. However, in this approach, the blockchain stores the execution trace of tasks and possibly also the data produced by these tasks, in a process, but it does not ensure that the execution of the collaborative process abides by a given process model. The process is executed within the (off-chain) BPMSs of the actors involved in the process, and these systems may perform tasks even if the current state of the collaborative process does not allow them to do so. Hence, this approach is not suitable when the goal is to implement an end-to-end collaborative business process in a way that benefits from the integrity of blockchain technology.

We observe that, except for [56], existing approaches for executing collaborative business processes on top of blockchain technology use the blockchain to record message exchanges or transactions. In addition, they use smart contracts to check or enforce that messages are exchanged in a way that is compatible with a collaborative process model. In other words, most parts of a business process are still executed in the BPMSs or other process-aware information systems of each business party, and the blockchain is used to record, monitor, and occasionally control interactions between the processes executed by each party. Also, existing approaches focus on a highly restricted subset of BPMN, comprising tasks, events, and `XOR` and `AND` gateways. In particular, these approaches do not support the execution of hierarchical process models (i.e., processes linked with sub-processes). Therefore, CATERPILLAR advances the state-of-the-art in the field by proposing a process execution engine with broad support for BPMN elements. Our approach enables the implementation and execution of collaborative business processes in which all critical components of the BPMS are hosted on the blockchain.

We also observe that existing approaches to blockchain-based business process execution mostly compile high-level process models into smart contracts that are deployed and executed on a blockchain platform [49, 56, 82, 94, 108, 150, 163]. These approaches, however, suffer from three limitations. First, they mostly focus on the control-flow perspective of process models. In other words, they do not handle process instance data (data perspective) nor the association between resources and tasks (resource perspective). The second limitation is that, due to their reliance on a compilation phase, these approaches do not support the adaptation of a process at runtime. In other words, a change in the model requires the generation and deployment of a new set of smart contracts (which is a costly operation), and existing process instances remain bound to the old version of the model. Third, compiled approaches often produce a significant amount of code (sometimes redundant) from the process models, leading to performance issues in blockchain environments in which the size of the code is limited. In CATERPILLAR, we tackle these limitations by adopting an interpreted approach as opposed to a compiled one. The key idea is to deploy a smart contract that can interpret BPMN models represented through a space-optimized data structure that can be modified at any time.

The work presented in [148] also adopts an interpreted approach. How-

ever, [148] focuses on the control-flow perspective (no case variables) and is limited to a small subset of BPMN elements (tasks and gateways with simplified execution semantics for join gateways). In particular, it cannot handle sub-processes, error events, and boundary events. In addition, this latter approach bundles together the interpreter with the data structures representing the process model into a singleton smart contract. Therefore, it suffers from the same flexibility issues as compiled approaches: Any change to the process model requires the deployment of a new smart contract (which is a costly operation), and existing instances remain tied to the previous smart contract. Finally, the approach in [148] requires that updates to the tasks in the process are performed by a central process owner, which is not suitable in scenarios in which there is no central trusted authority.

3.3. Flexibility in Collaborative Processes

In this thesis, we consider flexibility from two different viewpoints: resource and control-flow perspectives. Subsection 3.3.1 dives into existing works on the resource perspective and access control mechanisms to handle dynamic bindings of actors at runtime. Next, Subsection 3.3.2 focuses on the control-flow and the solutions regarding process variability, adaptability, evolution and looseness.

3.3.1. Resource Perspective: Access Control, Binding and Delegation Models

Traditional centralized access control systems are not suitable in collaborative processes in which trust is an issue. Typically, centralized systems require a third party in charge, which introduces risks of privacy leakage and constitutes a single point of failure [71, 135]. Blockchain and smart contracts are emerging as a solution to the problems of centralized systems in collaborative scenarios [135]. However, once deployed, a smart contract has no owner; i.e., undesired participants can access and call functions of it. Thus, blockchain-based systems require an embedded permission control mechanism to restrict who can perform each operation [91]. Existing blockchain-based access control solutions span different domains of application including general attribute-based auditable systems [95, 96], IoT devices [43, 93, 113, 116], data and resource sharing with emphasis in medical records [10, 41, 159, 169] and big data [48, 142]. These works use well-known access control methods existing in centralized systems, e.g., attribute-based, attribute-based encryption, RBAC, and fine-grained [135], to mitigate limitations regarding privacy, security, scalability, and performance. However, these existing solutions are either static or allow some dynamic operations that are mostly performed by administrators [28]. In addition, they do not consider the dynamic scenarios existing in collaborative processes in which the participants can be appointed/changed at runtime across different instances of a process.

In the domain of collaborative business processes, existing blockchain-based tools mainly focus on the control-flow perspective, as we discussed in Section 3.2. However, some early efforts are starting to consider the resource perspective too. Lorikeet [150] and the Blockchain Studio [100] tools implement static access control mechanisms in which roles are bound to accounts upon process instantiation. The tool presented in [147] supports three workflow resource patterns: direct allocation, role-based allocation, and separation of duties [16, 136], which are specified at design time. A method proposed in [125] by Prybila et al. allows dynamic handoffs of process instances between actors but does not support the specification and enforcement of permitted handoffs.

Existing rule-based automated resource allocation mechanisms [25, 26, 176] are not suitable in dynamic-collaborative processes, as these mechanisms assume that the rules for determining which resource will perform a given task can be determined upfront, at design-time. Similarly, other resource allocation solutions that use different techniques like process mining and social network analysis [83], machine learning [67, 68], and data mining [143], among others, rely on data which are not always accessible by all the participants in a collaborative process, thus making it subject to trust issues. In CATERPILLAR, 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 WS-BPEL [4], role binding is supported via *partner links* [11, 73]. A partner link is a variable that holds a reference to a service endpoint. This variable can be modified at any time during the execution of a process. This approach assumes that a single actor orchestrates the whole process 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 [78, 120]. A task-activity-based access control (TBAC) model, presented in [92], combines activities and dynamic permissions related to tasks in a business process. However, these approaches are not applicable in settings where a single actor does not determine the binding of actors to roles. Similarly, in [122], the parties can be replaced at runtime, but this always requires a central party serving as an intermediary.

Other studies have considered the problem of dynamic role binding in processes that are not orchestrated by a single actor. Robinson [132] extracts dynamic authorization policies from service choreographies, which are enforced locally by each party, but a central authority specifies all role bindings. BPEL4Chor [42] allows an actor to bind other actors to the roles it has control over. However, each role is controlled by a single actor. In other words, collaborative role binding is not supported; e.g., this approach does not support a scenario in which both the buyer and seller must agree on the actor who plays the role of the carrier. Also, BPEL4Chor does not support role re-binding. In [21, 157], 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 in which multiple actors must collaboratively agree on each decision.

3.3.2. Control-flow Perspective: Variability, Adaptation, Evolution and Looseness

Flexibility in the domain of process-aware information systems has been the focus of multiple research efforts. The latter is confirmed by many surveys published in recent years [9, 37, 106, 133, 144]. According to the taxonomy presented by Reichert and Weber [128], there are four major flexibility requirements: variability, adaptation, evolution and looseness. In the following, we discuss some relevant works related to each of those requirements.

Many works address the flexibility by variability, i.e., support for different variants of the same business process. The Provop approach discusses concepts on the design, modelling and management of process variants [62]. Ognjanovic et al. [115] consider stakeholder requirements, and Ayora et al. [8] propose a set of change patterns to configure and manage process families, respectively. Ellouze et al. present a version-based approach to model inter-organizational processes to deal with variability, adaptation and evolution [47]. Other empirical studies compare existing approaches for process variants, outlining their strengths and weaknesses [44], and providing criteria for selecting among the existing solutions [133].

No less attention has received the flexibility by adaptation, i.e., the possibility to deviate the execution flow for a given process instance temporarily. Among those, Nunes et al. [114] address the problem of runtime adaptation under the assumption that some known contextual elements can characterize unexpected situations. Klingemann [76] introduces flexible elements into the workflow specification to fulfil goals restricted by a controlled set of runtime conditions. Xiao et al. rely on re-usable process fragments which are dynamically related based on constraints and adaptation policies [170]. Pufahl et al. focus on the adaptation of batch activities [126]. On the automatic adaptation of workflows, the proposals in [102, 177] follows case-based reasoning approaches, and [104] relies on rules to specify exceptions and workflow adaptations. Others use planning strategies to automate business process reconfiguration at runtime [153] or to automate the construction of exclusive choices considering multiple paths under a set of specific variable conditions [63]. On service-oriented processes, the solutions described in [20, 127] automatically detects and solves adaptation issues, while [72] describes mechanisms to exchange web services in a Web service flow instance dynamically by extending WS-BPEL. Other solutions advocate for ad-hoc changes

to running instances on data-centric and data-aware [3], artefact centric [171] and activity-centric [65] processes, respectively.

Several other works focus on the flexibility by evolution, i.e., the permanent modification of a business process, which includes its active instances. Already in the '90s, two relevant research works envisioned that keeping the running instances consistency is one of the most challenging issues related to the workflow evolution [30, 80]. Those works also proposed a set of primitives to modify the workflow schema and migrate the running instances consistently. In this regard, Zhang et al. focus on the efficient migration of the running instances after the model modification [178]. Further, the solutions presented in [145, 180] provides insights about the management of different process model versions as a result of the workflow evolution. On service-oriented architectures, the work in [17] uses the case-transfer pattern on the evolution of inter-organizational workflows, while [149] considers change patterns on the evolution of event-driven processes. A more recent research in [81] transforms BPMN models into LNT process algebra and LTS formal models to check process changes and correct evolution errors.

Looseness is another of the flexibility requirements described by Reichert and Weber [128]. It refers to the ability to execute a business process whose control-flow is not fully specified or undefined at design time. Among the works addressing looseness, [2, 97] exploit the notion of worklets, i.e., self-contained sub-processes aligned to process tasks, which also support process evolution and adaptation at runtime. Worklets allows late-binding and late modelling, which are a typical implementation for flexibility by looseness. As such, other solutions also rely on the late-binding and late-modelling, e.g., on service choreographies and their orchestrations [164], to enforce patterns for executing collaborative tasks [36] and to handle temporal constraints to bind services [52]. The Declare language constitutes another significant contribution, which supports constraint-based representations of process models describing loosely-structured processes [121]. Similarly, Sadiq et al. introduced the concept of pockets of flexibility which allows execution of a process described by a loosely or partially specified model, whose full specification occurs at runtime [137].

However, these existing solutions, mostly focus on aspects like configuring, modelling, automation, validation and management of the flexibility requirements, putting aside scenarios in which trust is an issue, which is a crucial requirement for many inter-organizational processes. In other words, existing solutions mainly focus on how to update the process model consistently. However, partners in a collaborative setting may be suspicious about changes during process execution. Indeed, a partner may gain an unfair advantage by arbitrarily changing the model [99]. In that regard, the question of how untrusting participants can handle the updates of a collaborative process in a trusted way requires further research. In CATERPILLAR, we focus on flexibility in such scenarios that involve untrusting parties by using the blockchain technology.

In the blockchain setting, Prybila et al. [125] addresses the question of adapt-

ability. This approach caters to runtime adaptation, but it assumes that the process is not executed on the blockchain. The blockchain is used as a recording mechanism (recording the execution of tasks), as opposed to an enforcement mechanism (determining which tasks can be executed), as we do in CATERPILLAR. However, to the best of our knowledge, no other solutions have addressed the problem of control-flow flexibility on the blockchain.

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 [163]. Collaborative processes unavoidably involve tasks performed privately or requested by some party outside of the blockchain, and these are 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 among computers in the blockchain network.

This thesis, unlike existing approaches, introduces the concept of *flexibility by consensus*, such that untrusting participants can accept or reject requested updates to the process at runtime. To that end, we exploit the notion of late-binding [141] and worklets [2] in our role-binding and agreement policies to prevent inconsistencies that may result from process updates (see Chapter 6). More precisely, we follow an approach of flexibility by underspecification [141] or looseness [128], in which the specification occurs at runtime and may vary with each instance [137]. However, the interpretation-based engine additionally supports the variability, adaptability and evolution requirements. The later three flexibility requirements are unsupervised, i.e., the participants of the process are responsible to keep the updated model consistent (see Section 5.1.2)

4. CATERPILLAR: A BLOCKCHAIN-BASED BUSINESS PROCESS EXECUTION ENGINE

In this chapter, we focus on the research question **RQ1**: *How can the high-level abstractions of BPMSs be combined with the capabilities of blockchain technology to support the execution of collaborative business processes between mutually untrusted parties?* To that end, this chapter focuses on the design and implementation of the CATERPILLAR system. Specifically, in this chapter, we follow a compiled approach to execute blockchain-based processes. As disused in Section 1.1.2, compiled approaches are suitable in scenarios in which the flexibility is not a requirement, or in blockchain platforms which required no fee payment for the deployment of the contracts, e.g., consortium blockchains like Hyperledger Fabric. Also, these approaches promoting the immutability of smart contracts as a source of trust remains among the most widely used [99].

The chapter is structured as follows. Section 4.1 introduces a business process model written on the BPMN standard that we use as a running example. Next, Section 4.2 describes the architecture of the CATERPILLAR system following a compiled approach, i.e., the process models are translated into immutable smart contracts. Section 4.3 delves into details of the compilation from BPMN into solidity. Then we describe the implementation and evaluation of the system in Section 4.4 before Section 4.5 concludes the chapter.

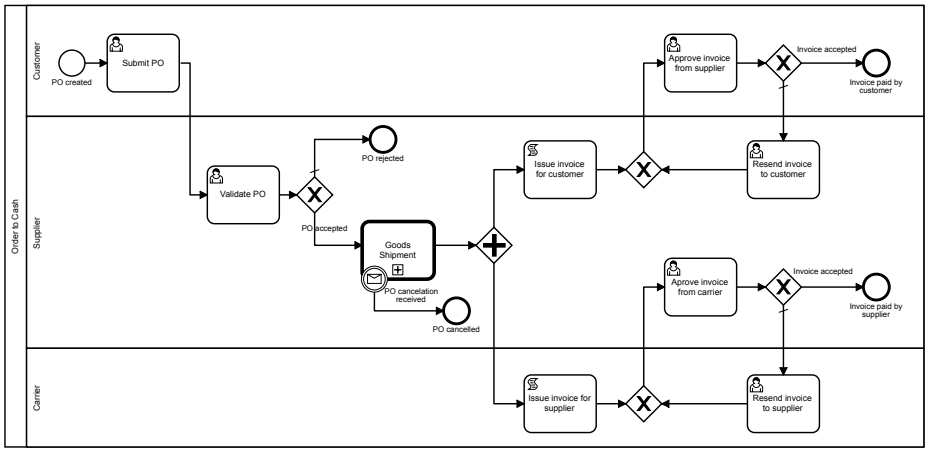
4.1. Running Example

In the following, we introduce a running example that will be used throughout the chapter. As discussed in the introduction of the thesis, CATERPILLAR takes as input single pool process models, and not choreographies nor collaboration diagrams. Accordingly, the participants must agree in the set of flow elements describing a joint a process model [58, p. 143]. Figure 9 presents the business process to illustrate concepts and CATERPILLAR's components. For convenience, the business process is modelled into two separate diagrams: an ORDER TO CASH root process model and a reusable GOODS SHIPMENT sub-process, which is called from the root process model. The model includes three participants, Customer, Supplier and Carrier, represented by the three lanes in the model, instead of pools as for collaboration diagrams. Similarly, traditional message exchanges are replaced by user tasks; through them, participants can check-in and check-out information.

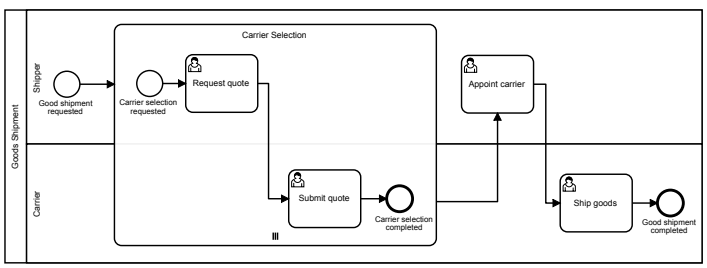
The process starts with a none start event. The latest is a requirement as CATERPILLAR only supports so-called *explicit instantiation*, i.e., it does not support implicit instantiation via start timer events or start message events.

The process execution starts with the user task `Submit PO`.[1] A user task adds

[1] We are using PO as an abbreviated form of purchase order.



(1) Root process: Order-to-Cash



(2) Sub-process: Goods-Shipment

Figure 9: Running example: An order-to-cash process (1), with a shipment sub-process (2)

a work-item into a stakeholder's worklist. Therefore, the user task `Submit PO` is intended to let a stakeholder enter the details of the purchase order. Next, the case continues with the user task `Validate PO`. This task intends to allow a stakeholder to check the validity of the PO, the existence of the goods in the warehouse, to later submit a decision over the PO (e.g., PO is accepted or rejected). Next, the execution of the process proceeds reaching an exclusive gateway, which selects one out of the two possible execution paths based on a predicate formulated over the process data, i.e., the decision taken by the stakeholder in regards of the validity of the PO. When the PO is rejected, the execution flow will reach the end event `PO rejected`, ending the execution of the process instance. On the other hand, if the PO is accepted, the execution flow will reach the call activity `Goods Shipment`. Executing a call activity implies the instantiation of the GOODS SHIPMENT sub-process. The boundary event attached to the call activity GOODS SHIPMENT indicates that the supplier has the right to cancel the order so long as the goods have not yet been dispatched.

The execution of the sub-process GOODS SHIPMENT eventually reaches a

point where multiple instances of the sub-process CARRIER SELECTION are created. The intuition behind is that each instance of the sub-process implements the interaction between supplier and carrier companies to get a quote.[2] When the carrier companies submit all the quotes, a clerk on the supplier company selects one carrier based on the quotes and organizes the shipment. Then, when the user task SHIP GOODS finishes, the execution flow passes back to the root process. The process instance proceeds by activating the two parallel paths to process the payment of the shipment by the supplier and the payment by the customer. At this point, the process includes two script tasks, i.e., rounded rectangles with a folded paper-like icon, which represent Solidity scripts to issue invoices. Once the invoices are issued, the customer and supplier should pay their corresponding invoices. Note that the process considers the possibility of reissuing the invoices, e.g., when one invoice is wrong. The overall process ends when the parallel paths reach their corresponding end events.

4.2. Architecture of the Caterpillar System

The principles described in the introduction of this thesis (Section 1.2) guided the design of CATERPILLAR's architecture, which is organized into three layers as shown in Figure 10. The layer at the bottom will be referred to as the "On-chain Runtime and Storage" layer. Specifically, the "On-chain Runtime" refers to a set of smart contracts that includes housekeeping support code, e.g., process instantiation, as well as process-specific code, e.g., control-flow, process data, etcetera. The on-chain parts are replicated across all full nodes of the blockchain network, e.g., the public Ethereum network. "Storage" refers to the "Ethereum log", which serves to externalize process events and data, and the "Process repository", which keeps compilation artefacts and the like.

In the middle, the "Off-chain Runtime" layer includes a set of tools to compile, deploy and monitor business processes on the On-chain layer. Specifically, the Off-chain runtime includes a BPMN compiler, a deployment mediator, an execution monitor and an event monitor. Note that these Off-chain runtime components can be hosted by each actor in a collaborative process separately. In particular, all actors do not need to use the same event monitor or execution monitor. Following our design principles, the state of all process instances is stored on-chain, and all the decision points are evaluated on-chain. Hence, if one of the event monitors tries to execute an event at the wrong moment, it would be rejected by the corresponding smart contract. Similarly, each actor can host its own BPMN Compiler to generate the smart contracts implementing a process model – and in this way, they can cross-check the contracts deployed on the blockchain. Also, each actor

[2]The demonstration process included in CATERPILLAR's code repository creates a fixed number instances of the CARRIER SELECTION sub-process. The number is specified with a process variable and is set to two by default.

can use their own deployment mediator, though, on end, only one of them should deploy a given process (the others can check the deployed contracts).

Finally, the top-most layer comprises a set of components for editing executable process models, packaging process configurations (e.g., adapting control-flow code with components implementing different resource management schemes, etcetera.), and to monitor the execution of process instances/cases.

Compared to the WfMC and to the more recent BPMS-RA reference architectures [66, 124], the component in the CATERPILLAR architecture can be classified as follows. (i) *Process Definition* comprehends the "Modeling Panel" and "Process Repository", as they serve to define and to store all the information regarding the high-level process models. (ii) *Workflow Enactment Service* encloses the "BPMN Compiler", "Deployment Mediator", "Runtime Registry", "Contract Factories", "Workflow Handler". They are responsible for the compilation, creation, management and execution of the workflow instances. (iii) *Administration and Monitoring Tools* relates to the "Execution Monitor" and 'Worklist Handler". They handle resource and user access control management as well as the execution of enabled tasks. (iv) *Workflow Interoperability* includes "Event Monitor", "Service Bridge" and "Ethereum Log", as they support the interactions between the on-chain components with external systems. Specifically, to request an external interaction and event is placed in the "Ethereum Log", and later processed by the "Event Monitor" (off-chain). (v) *Workflow Client Functions* comprises the "Modeling Panel", "Configuration Panel" and "Execution Panel", which enable the interaction of the end-users with the CATERPILLAR's off-chain and on-chain components. Due to the separation between on-chain and off-chain components in CATERPILLAR, some of the components may fall into several classifications. Note that, in CATERPILLAR, the participants always share the on-chain components. However, they may either implement a different or share the same off-chain components, i.e., the top layers REST API and Web Portal in the architecture. According to that, the interactions among components may vary. Below, we provide a more detailed description of the components in each one of these layers.

4.2.1. On-Chain Runtime and Storage

The dashed rectangles in Figure 10 divide the bottom layer of the architecture into two parts. On the right, the "On-chain Runtime" components running on the Ethereum blockchain platform store and support the execution of smart contracts that fully encode a set of process models. The events generated by these smart contracts are recorded in the blockchain platform's log, which is accessible from outside the blockchain. On the left, a "Process Repository" is used to keep data received, produced, and required by CATERPILLAR to execute the process models deployed on the blockchain. Below we discuss these two parts in turn.

Process Repository and Ethereum Log

A smart contract has no way of calling an external resource directly. To cope

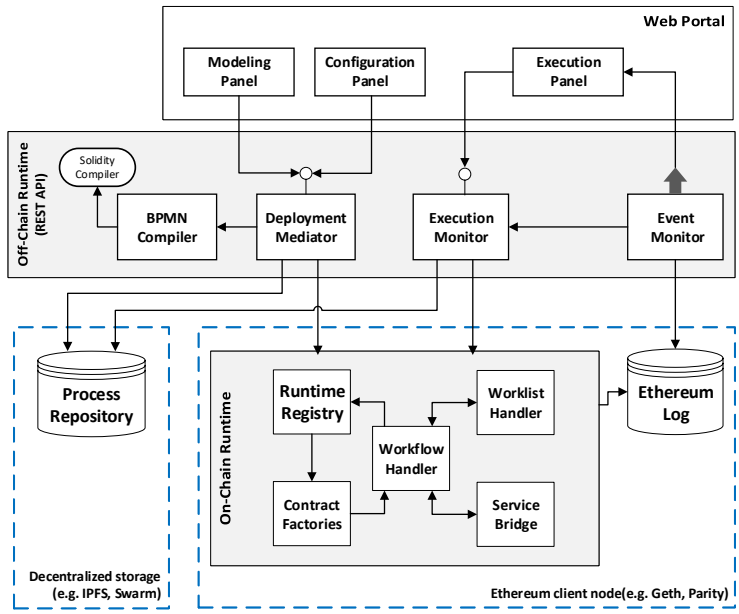


Figure 10: The architecture of CATERPILLAR: the compilation-based engine.

with this limitation, CATERPILLAR publishes Solidity events on a log that is stored inside each full blockchain node, which external programs can read. Thus, external programs and users can react to the events and submit responses in the form of transactions to the blockchain. Software that follows this style of interaction is often referred to as an oracle [22]. An oracle serves as a proxy to mediate between smart contracts and external applications and often includes another contract dedicated to it. In CATERPILLAR, the "Ethereum log" (right-hand side of 10) provides the medium for communication between off-chain and on-chain components. Specifically, when a transaction is included in the blockchain, CATERPILLAR emits and write an event in the log to notify external components that some updates took place, e.g., to announce that some task became enabled; thus participant can execute it. This mechanism is used to manage interactions between the process-related smart contracts generated by CATERPILLAR, and external resources such as software services, as discussed in Section 4.3.

The "Process Repository" (bottom, left-hand side of Figure 10) stores and provides access to compilation artefacts, including the BPMN process models, the Solidity code generated from them, and additional metadata to link the generated Solidity code to elements of the BPMN models. This metadata is used during the deployment of the Solidity smart contracts (when creating a new process instance) and also to link the state of a running process instance to the corresponding model. The Process Repository can be implemented, for example, on top

distributed storage platforms like the InterPlanetary File System (IPFS).[3] We note that the compilation artefacts in the process repository could be stored directly on the blockchain as a stream of bytes inside a smart contract. However, this alternative approach would entail a high storage cost, given that the storage is immutable for the lifetime of the blockchain. IPFS provides an alternative decentralized approach to storing the compilation artefacts at a lower cost while producing an immutable and unique cryptographic hash key that uniquely identifies each of the compilation artefacts, and ensures the absence of manipulation (albeit without guaranteeing availability).

The process-related smart contracts generated by CATERPILLAR store the following data on-chain for each process instance: (i) the state of the process instance from a control-flow perspective, in order to determine which tasks are enabled/started; [4] (ii) the data that needs to be given as input to the tasks of the process; and (iii) the data required to evaluate the conditions in the decision gateways. Beyond these minimum requirements, a developer may specify additional variables in the BPMN model, in which case these variables are also stored on-chain, but this is optional. As an alternative and in order to strike a trade-off between costs and being tamper-proof; it is possible to store a link to the full data objects on-chain (and possibly also a hash code for verification) and keep the full data off-chain, or to keep the full data objects in IPFS, for example by appending it to the CATERPILLAR's process repository. This latter approach can also be applied when the volume of data that needs to be given as input to user tasks and service tasks is too large. In other words, the full data objects required by these tasks can be stored off-chain or in IPFS, so that only a link and a hash code need to be stored on-chain. A discussion on the performance, availability, and security properties of these alternative design decisions for data storage can be found in [173].

On-Chain Runtime Components

The "On-chain Runtime" consists of five components as shown in Figure 10. First, the "Workflow Handler" comprises the set of smart contracts generated by CATERPILLAR from the input BPMN models to handle the control-flow of the process models. The next two components named "Worklist Handler" and "Services Bridge" consist of smart contracts that enable the interaction with external applications and to validate any data checked-in into to the process instances, thus managing the interactions with external applications and users as specified by user tasks and service tasks in the BPMN model. The Worklist handler is responsible for managing user tasks (i.e., tasks to be performed by end-users in BPMN). At the same time, the Service Bridge handles service tasks in BPMN, i.e., programmatic interactions with external applications exposed as services. The Worklist Handler

[3] https://ipfs.io/

[4] The life-cycle of an element in CATERPILLAR is further discussed in Section 4.3.2 (Figure 15). Overall, enabled activities to be executed by a participant, through an oracle, are marked as started. Then, when the corresponding transaction is included in the blockchain, the activity is marked as completed. Thus, an activity transits from the states enabled, started and completed.

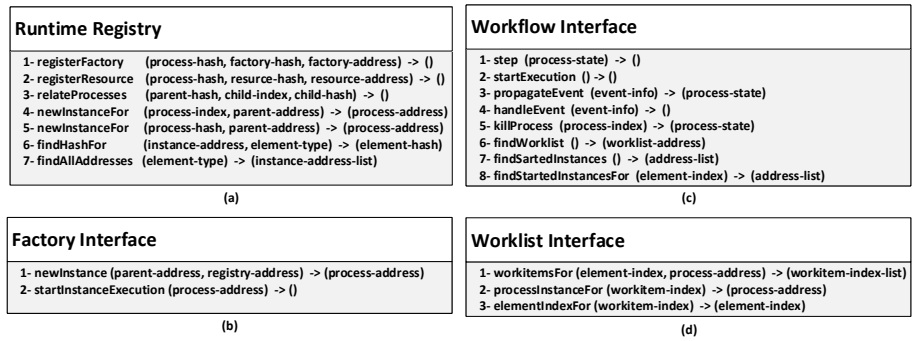


Figure 11: Interfaces with their operations in the smart contracts managed by the compilation-based engine of CATERPILLAR.

and Service Bridges are implemented similarly: they consist of a smart contract that acts as the mediator for forwarding a request (via Solidity events) and receiving the corresponding response (via a contract function call). CATERPILLAR provides a simple and generic implementation of the Worklist Handler, which keeps track of work items that are enabled, started, or completed. In contrast, the "Service Bridge" is a set of smart contracts that must be provided together with the BPMN process model, because CATERPILLAR cannot generate it as the details of these contracts are dependent on the services to which the bridging is made. The bridge contracts need to implement a program interface specified by CATERPILLAR.

The component "Contract Factory" includes a set of contracts that serve to instantiate the smart contracts associated with a BPMN model as required: it instantiates the smart contracts implementing the "Workflow handler" and "Worklist handler" of the root process, binds them and then fires the execution of the process instance.

The fifth on-chain runtime component, namely the "Runtime Registry", is a smart contract that keeps track of process instances (addresses of deployed "Workflow handler" smart contracts) and their relation with other smart contracts in CATERPILLAR's on-chain runtime. The main functionalities of the "Runtime Registry" are shown in Figure 11(a). The suffixes on the name of the parameters hint at the location of the underlying artefact. When the artefact refers to metadata stored in IPFS, then a hash is used as an identifier. When the artefact refers to smart contracts, then the identifier corresponds to the address of the deployed contract. When parameter name uses an index as the identifier, then a level of indirection (e.g., a Solidity mapping) is used.

Figure 11(b)-(d) also introduce the interfaces implemented by the three typical contracts produced by the "BPMN Compiler". The functions in *Workflow Interface* can start the execution of an instance, perform internal steps updating the process status, throwing and catching events, finish running instances, as well as

finding information of related resources and contracts, e.g., invoked through call activities. On the other hand, the *Worklist Interface* provides functions to access the information of work-items. A work-item, identified by an integer index, includes the address of the related control-flow contract and the index of the activity to be performed by a process participant. Also, contracts implementing both interfaces above include a set of functions relying on the elements extracted from process models (see Section 4.3 for further details). Finally, factories must implement two methods. The first function instantiates the corresponding process, taking as input the addresses of the "Runtime registry" and the contract which is responsible for creating the new instance (zero if an off-chain component executes the operation). The second function forces that process executions must be explicitly started, not when creating the new instance in the blockchain. CATERPILLAR automatically executes some elements internally at the moment they are enabled as explained in Section 4.3. Therefore, any contract instantiated from an element, e.g., a call activity, must be deployed before starting the execution when the corresponding element has been reached.

Before creating new instances of a process, the corresponding factory contract must be deployed on Ethereum. The address of the corresponding process factory must be stored/associated with an identifier to a process model in the repository. The factory of this model-factory mapping can be updated at any time using the function `registerFactory`, which allows a process model to be implemented using different strategies. Besides, as interfaces offer no information about the concrete implementation of the factory, an identifier to recover such information from the repository must be provided. A similar approach applies to link a process with the contracts supporting the interaction of participants, i.e., worklist for humans or services for information systems, using the function `registerResource` in both cases. Relations of processes activities and other BPMN elements linked/mapped into smart contracts are also stored, using the function `relateProcess`. In this case, the parent refers to the process containing the element with the link. Unlike previous relations, here the parent will instantiate a child given its index, but beforehand the referred contract must be deployed. Thus the corresponding identifier is required for the registration.

CATERPILLAR forces the process instantiation to be made through the "Runtime Registry". To that end, two alternatives are available described by the functions `newInstanceFor`. An off-chain component must provide the identifier of the process as shown for the first variant of the function. In this case, the parent address is zero because CATERPILLAR prevents external actors from creating process instances linked to other process instances – this mechanism is internal to CATERPILLAR. In other words, external actors only can instantiate root processes, never sub-processes directly. Indeed, a parent process has to create instances of a child given the integer index of the BPMN element linked to the corresponding contract. In both cases, the "Runtime Registry" verifies if the required contracts were deployed and instantiated (e.g., factory, worklist, service), and then chooses

the factory accordingly. Once created the instance, its address is published in the event log of Ethereum to notify external actors the sub-process instance creation and how it can be reached. Besides, the "Runtime Registry" keeps a record of this new address that is associated with the process identifier.

The "Runtime registry" also provides methods to retrieve the identifier of a process running at a given address. Note that this identifier serves to find and check the information related to the process and its compilation metadata from the repository. Besides, the list of all the addresses created for any category of contracts can be recovered from the registry. In Figure 11 (a) the parameter *element-type* refers generically to the different categories of contracts, e.g., factories, worklist, service and process models. In summary, the "Runtime registry" provides full control of the processes deployed by CATERPILLAR; the history of process executions, active instances, relations between contracts, and the metadata required to create new instances can be retrieved.

4.2.2. Off-chain Runtime

The off-chain runtime component of CATERPILLAR provides a service-oriented layer that allows external applications to interact with the on-chain components and the repository. The off-chain components enable external applications to compile process models into Solidity smart contracts, to deploy the smart contracts, to query the status of process instances, and to register the execution of tasks associated with active process instances. Accordingly, the off-chain runtime consists of three modules: a BPMN compiler (which uses a Solidity compiler), a deployment mediator, and an execution monitor. The latter component relies on a fourth component, namely the Event Monitor, which listens for relevant execution events from the Ethereum log.

The runtime off-chain components are optional, i.e., the parties can directly invoke the smart contract transactions without going through CATERPILLAR's off-chain components, or they can implement their private runtime. The CATERPILLAR on-chain components ensure that, should an off-chain component be tampered, this would not affect the integrity of the process execution recorded on the blockchain, since the transaction corresponding to a task is executed if and only if the current state of the corresponding process instance allows so.

However, the tampering of an off-chain component may affect what the parties observe (e.g., the notifications they receive). For example, if a fake carrier tampers the "Event Monitor" to notify that the goods can be shipped when the actual task has not been performed in the blockchain. Then, that carrier may be able to pick up the products, i.e., if the supplier did not check the actual event in the blockchain. Note that dispatching a product happens in the real-life and not in the on-chain environment. The involved parties can mitigate this vulnerability by introducing an additional secured software component that queries the blockchain to check that a task is enabled before executing it. Then, if a supplier receives a

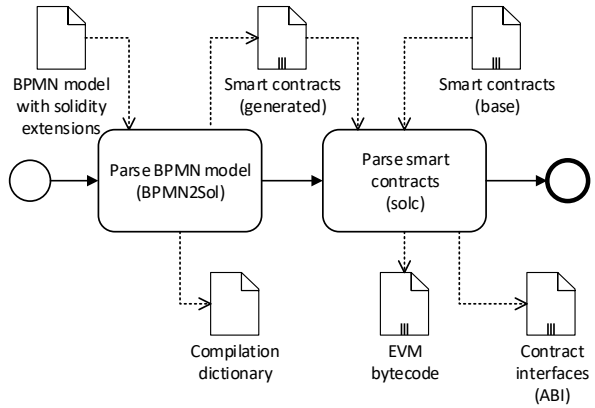


Figure 12: Caterpillar's compilation process

notification from the off-chain component indicating that the products should be shipped, the additional secured layer should check that the corresponding task is enabled via the on-chain components before performing this task. In this way, the potential tampering of the off-chain components would be detected, and the goods would not be shipped.

The Compiler

The first off-chain component, the BPMN compiler, is responsible for compiling BPMN process models into smart contracts. This compilation is done in two steps (cf. Figure 12). First, the BPMN process model is compiled into a set of Solidity smart contracts plus additional metadata, called the *compilation dictionary*, which is used later for monitoring purposes. The compilation dictionary is a data structure that includes information to map the elements in the BPMN model to the generated code. This information includes the name of the contract method associated with any activity, a unique integer index assigned to each element, as well as the respective element type.

In the second step of the compilation process, CATERPILLAR puts together the set of smart contracts produced in the first step and, if applicable, the set of already existing contracts such as the ones corresponding to the interfaces of service tasks and call activities. These Solidity smart contracts are passed to the solidity compiler which produces EVM bytecode and ABI[5] definitions that are used for deploying the smart contracts to Ethereum. The ABI definitions are further used by CATERPILLAR's off-chain runtime or any other third-party applications to interact with the deployed contracts. Artefacts involved in the compilation process, that

[5]ABI (Application Binary Interface) is a JSON-based description of the list of the public methods implemented by a Solidity smart contract as well as their signature. Several language bindings use the ABI definition, e.g., web3js in the case of JavaScript, to enable the interaction with smart contracts deployed into Ethereum.

is, input BPMN models, solidity contracts, compilation dictionary, EVM bytecode and ABI definitions are stored in the "Process Repository".

The CATERPILLAR's compiler produces three smart contracts from an input model if it is flat (i.e., contains no sub-processes): the workflow, worklist, and factory contracts. The first one implements the data and control-flow perspectives; the data perspective is embedded as part of the control-flow implementation, although. The second contract, the worklist, handles the execution of work-items by stakeholders named and serves to send/receive the process data. The third contract, the factory, provides a default mechanism to create instances of the process. As stated above, service tasks involve interactions with information systems running outside of the blockchain, and require other smart contracts similar to worklists. Additionally, some modelling elements, such as multi-instance activities and call activities, are implemented in separate contracts. Note that, if a process contains at least one sub-process (i.e., embedded or linked to some call activity), then a relation parent-child is implicitly established. Considering that each sub-process can, in turn, have a set of children, then the sub-processes define a hierarchy.

Deploying Smart Contracts

Once the compilation process finishes with the resulting metadata stored in the repository, the root contract can be instantiated from the "Deployment Mediator". From this point, the contracts' identifiers are the hashes produced by the repository. Such hashes also serve as the key to access the compilation metadata. Figure 13 shows the steps performed by CATERPILLAR when deploying a contract, i.e., before creating the first instance. Initially, any parent-child relations are updated in the registry. Next, for any contract in the process hierarchy, the corresponding factories and resources (i.e., worklists and services) are instantiated and associated accordingly in the registry. Note that instantiating a contract off-chain requires the bytecode produced by the solidity compiler. On the other hand, calling or invoking a function requires the contract ABI in conjunction with the address of a running instance. Finally, the root process is instantiated and subsequently started. In contrast to factories and resources that must be explicitly instantiated,[6] the workflow contract is instantiated through the "Runtime Registry" which also updates itself with the newly-created addresses.

As hinted in Figure 13, most of the steps are optional as CATERPILLAR allows lazy operations. In other words, registrations must be performed before elements involving an interaction are reached in the control-flow, but there is no specific moment to register such information. For example, in the process model presented in Figure 9 it is mandatory to register the contract produced for the process ORDER TO CASH as root to allow the execution of any of its tasks by an off-chain component. Besides, the factory is required to create the instance and starting the

[6]There is also possible to register instances of factories/resources created by another runtime component.

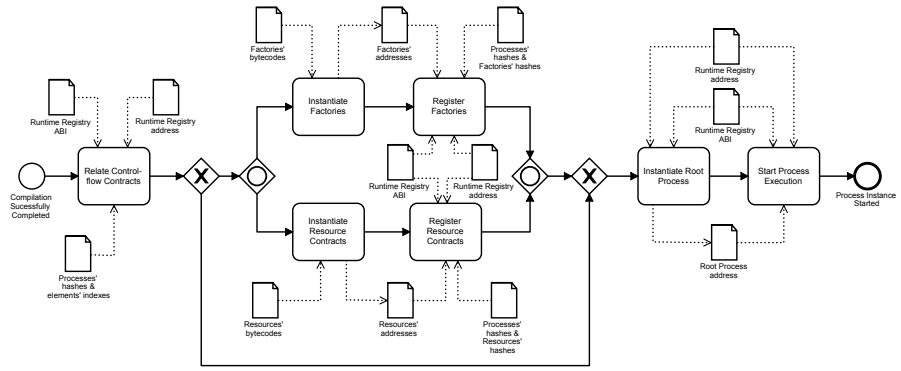


Figure 13: Process Instantiation through Caterpillar's compilation-based engine.

execution, upon which the user task `Submit PO` is reached in the control-flow. In order to execute that task, the corresponding worklist is also required. The contracts related to the call activity GOODS SHIPMENT as well as the multi-instance CARRIER SELECTION can be registered at any moment before reaching the respective elements in the control-flow. However, it is advisable to register any involved interaction before instantiating a contract to avoid unexpected runtime errors resulting from reaching unlinked elements during the execution.

Querying Process State and Executing Tasks

In order to inform a user which activities she can perform, we need to query the process state. More specifically, we need to derive the set of started user tasks from the information in the smart contracts. Once a process model has been deployed into CATERPILLAR's runtime for execution, any instance of it evolves from its initial state, executing a subset of the underlying activities until no activity is found to be a candidate for execution or currently being executed. Following this intuition, the relevant part of the state of a process instance at a given point in time can be identified with the subset of activities that are in the state *started*. Note that any activity that is *enabled* during the execution is *automatically started* by CATERPILLAR, thus simplifying the querying task. Furthermore, a process instance may give rise to more than one contract instance, e.g., from a multi-instance activity or a call activity. Nonetheless, it is the address of the instance contract related to the root process, which would be used for querying the state of the process instance as a whole. To that end, CATERPILLAR traverses the hierarchy of smart contracts implementing the behaviour of a process instance and collects the set of started activities therein. It is easy to see that the hierarchy always corresponds to a tree. Henceforth, querying process instance state can be done via traversal of such a tree of contracts.

At any time, an external entity can query the state of a given process instance, i.e., the set of activities that are currently active (or executing) for that particular instance, through the "Execution Monitor". Algorithm 1 illustrates CATERPILLAR's off-chain runtime approach for querying the state of a process instance.

Given the address associated with a process instance, the algorithm performs a depth-first traversal over the tree representing the hierarchy of contracts, collecting the information of the started elements.

Algorithm 1 Querying a process instance state

```
 1: function INSTANCESTATEFOR(process_address, registry_address, registry_ABI)
 2:     runtime_registry = Contract.at(registry_address, registry_ABI)
 3:     work-items, service_tasks, PENDING ← ∅, ∅, ∅
 4:     PUSH(PENDING, process_address)
 5:     while PENDING ≠ ∅ do
 6:         instance_address ← POP(PENDING)
 7:         process_hash = runtime_registry.findHashFor(instance_address, 'PROCESS-CONTRACT')
 8:         dictionary ← FINDDICTIONARY(Repository, process_hash)
 9:         instance_contract ← FINDCONTRACTFOR(runtime_registry, instance_address, 'PROCESS-
    CONTRACT')
10:         worklist_address ← instance_contract.findWorklist()
11:         worklist_contract ← FINDCONTRACTFOR(runtime_registry, worklist_address, 'WORKLIST-
    CONTRACT')
12:         for element_index ∈ instance_contract.startedActivities() do
13:             case dictionary.type_of(element_index ) of
14:                 work-item:
15:                     work-items ← work-items ∪ worklist_contract.work-itemsFor(element_index, in-
    stance_address)
16:                 SERVICE:
17:                     service_tasks ← service_tasks ∪ {(instance_address, element_index) }
18:                 SEPARATE_INSTANCE:
19:                     for subinstance_address ∈ instance_contract.findStartedInstances(element_index ) do
20:                         PUSH(PENDING, subinstance_address)
21:             end case
22:     return (work-items, service_tasks)

23: function FINDCONTRACTFOR(runtime_registry, instance_address, contract_type)
24:     element_hash ← runtime_registry.findHashFor(instance_address, contract_type)
25:     contract_abi ← FINDABI(Repository, element_hash)
26:     return Contract.at(instance_address, contract_abi)
```

Initially, the input address, i.e., process_address, is pushed into the stack represented by the variable PENDING (line 3) before entering the while loop. Besides, the address where running the "Runtime Registry", as well as its ABI, are required to invoke some functions. So, they are provided as input. In line 2, the algorithm calls the function `Contract.at` to instantiate a contract wrapper (e.g., contract object for solidity contracts according to the vocabulary used by the web3.js JavaScript library), which will be later used to call functions of the solidity contract running in the blockchain. Note that a similar approach is used to recover wrappers from processes and worklists, in the function named `findContractFor` in lines 27-31. However, in this case, firstly, it is necessary to recover the hash and ABI from the registry and repository, respectively.

The traversal is implemented as a while loop (lines 5-24). At each iteration, one contract address is processed. In lines 7-8, the algorithm retrieves the compilation dictionary, which will be later used for determining the type of process element being processed. The address of the associated worklist is retrieved (line 10) and used later to obtain the worklist wrapper. The `for` loop in lines 12-24 iterates over the set of started elements that CATERPILLAR on-chain runtime reports

Algorithm 2 Executing a task by an external actor

```
1: function EXECUTETASK(worklist_address, work-item_identifier, input_parameters, runtime_registry)
2:     worklist_contract ← FINDCONTRACTFOR(runtime_registry, worklist_address, 'WORKLIST-CONTRACT')
3:     element_index = worklist_contract.elementIndexFor(work-item_identifier)

4:     worklist_hash ← runtime_registry.findHashFor(worklist_address, 'WORKLIST-CONTRACT')
5:     dictionary ← FINDDICTIONARY(Repository, worklist_hash)
6:     function_name ← dictionary[element_index].function_name

7:     EXECUTEFUNCTION(worklist_contract, function_name, input_parameters)
```

for a given address, using the corresponding contract wrapper.[7]

In the CATERPILLAR's off-chain runtime, there are two types of activities for which the "started" state is externally visible: those associated with work-items (i.e., user tasks, receive tasks and message events) and the service tasks. Lines 18-21 will be executed when a BPMN element is associated with a separate contract instance, e.g., if the element found in the started state is a call activity. In such cases, the `for` loop in lines 19-21 iterates over the set of contract instances associated with the BPMN element at hand, pushing their corresponding contract addresses into PENDING for further processing. The algorithm returns two lists, one with the indexes of the work-items of the started user tasks and a second list with the started service tasks which are represented by a pair enclosing the task index and the corresponding instance address. Also, determining which function is associated with an element requires to query the dictionary.

Querying other useful information, such as the deployed process models or the addresses of running instances, is relatively trivial. Such queries are made by the "Execution Monitor" either by calling the corresponding functions in the "Runtime Registry" or by checking the process repository.

The "Execution Monitor" also allows executing the started activities by external actors with the required access authorization. Nevertheless, such execution must start from the worklist contract, that validates the interaction before redirecting the call into the workflow contract. Algorithm 2 illustrates the steps needed to perform user tasks. To that end, the address of the corresponding worklist, the work-item index, as well as the values of the input parameters, must be provided. Note that, although it is not explicit in the Algorithm 1, the information about worklists and parameters, required to execute started tasks, is provided when querying the process status.

Initially, in Algorithm 1 the function `findContractFor` in lines 27-31 retrieves the interface for the worklist contract used later to find the index associated with the corresponding BPMN element, i.e., the user task. Next, the compilation dictionary provides the name of the function to be executed in the worklist, as it

[7] As described later in Section 4.3, the set of started activities is represented with an integer, manipulated as a bit set and the line 12 in the algorithm conceptually captures the iteration over such set. Henceforth, each BPMN element in that set is represented by a bitmask, which we refer to as element_index in the algorithm.

shows the line 7. In this case, as the function name is just a `String`, the execution occurs without invoking the function on the contract wrapper. Instead, a sort of interface, e.g., provided by web3.js JavaScript library, is used to resolve such execution.

Finally, given that the interactions with Ethereum occur asynchronously, the "Event Monitor" listens for low-level events. This component processes the Ethereum event log to determine when to push notifications either to the "Execution Monitor" or to any external application, e.g., when a task has completed or after executing a new work-item.

4.2.3. Web Portal

The CATERPILLAR's Web Portal exposes the functionality of the off-chain runtime component to end users (e.g., process administrators and process workers) via a form-based user interface. The Web portal is structured into three panels: "Modeling", "Configuration" and "Execution" (cf. Figure 10).

The "Modeling panel" allows the user to draw the BPMN models that are deployed later into the blockchain. Besides, it is possible to import and edit models produced by another tool if they comply with the BPMN standard. In both cases, the models are typically enriched with the Solidity snippets which later are embedded into the smart contracts produced. Note that, in case of errors in such Solidity code, the compilation fails; after fixing the errors, the modeller can try to recompile the model. As indicated in Figure 10, the models created in the "Modeling panel" are deployed to the blockchain through the "Deployment mediator" if the compilation succeeds.

The "Configuration panel" supports introducing new relations or updating the links defined for process models already deployed on the blockchain. For example, it is possible to import and instantiate a smart contract of a worklist produced not by CATERPILLAR (but following the structure expected by the tool) that are associated later with a deployed process. This component is also the entry point to deploy and instantiate the contracts which implement the interactions with the service tasks. Besides, changing the links of a process to another, e.g., through a call activity, is also supported. In other words, the "Configuration panel" provides a set of functionalities to introduce and update the information in the "Runtime Registry" at any time during the process execution.

The "Execution Panel" interacts with the "Execution monitor" to retrieve all the information about deployed models, running instances and also allow executing tasks by stakeholders. For example, a user can access the list of deployed processes to choose the desired one. Then, all the information stored in the "Process repository" can be checked as well as the list of running instances can be retrieved, in this case from the "Runtime Registry". Besides, given one instance of a selected process, a user can visualize its state. Moreover, the started user tasks can be executed, more specifically, the "Execution Panel" will generate a Web form

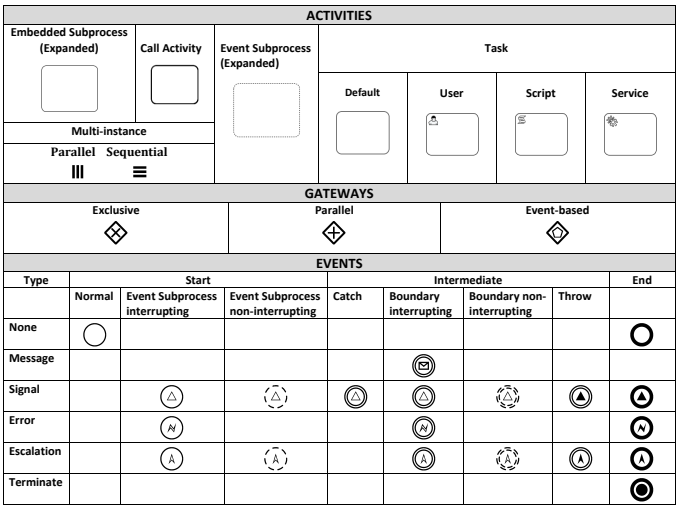


Figure 14: BPMN elements supported by CATERPILLAR.

to let users provide input data to a process instance which will be validated by the corresponding worklist. Finally, the "Execution panel" receives notifications from the "Event Monitor" to keep track of the transactions included in the blockchain and update the visualization accordingly.

4.3. Compiling BPMN into Solidity Smart Contracts

The CATERPILLAR's compiler takes as input a BPMN process model annotated with Solidity code snippets. The input BPMN model may contain user tasks, service tasks, script tasks (with Solidity scripts), gateways (exclusive, parallel, event-based and complex), events (none, terminate, message, signal, error and escalation), call-activities, embedded sub-processes, event-sub-processes and multi-instance activities (parallel and sequential), as summarized in Figure 14. It also supports message events attached to the boundary of an activity (i.e., interrupting and non-interrupting message boundary events).[8]

Below, we discuss how CATERPILLAR generates smart contacts from a BPMN process model. We start by discussing the handling of variables and interactions with external resources via user and service tasks. We then discuss how the control-flow (sequence flows and gateways), sub-processes, and events are encoded in the generated smart contracts.

4.3.1. Process variables and external resources

Listing 1 shows an excerpt of the smart contract that CATERPILLAR generates from the ORDER-TO-CASH process in Figure 9. This contract is called the *main*

[8]For a detailed explanation of the BPMN elements handled by CATERPILLAR, we refer the readers to Section 2.2 of this thesis.

```
1  contract OrderToCashProcess {
2    // == RESOURCE MANAGEMENT ==
3    address worklist;
4
5    // == DATA-FLOW PERSPECTIVE ==
6    enum POStatus {PENDING, ACCEPTED, REJECTED, CANCELED, CLOSED}
7    bytes32 sku;
8    uint quantity;
9    uint price;
10   POStatus status;
11   ...
12
13   function ValidatePO_Complete(POStatus decision) external {
14     require(msg.sender == address(worklist) && /* control-flow
           validations */);
15     require(decision == POStatus.ACCEPTED || decision == POStatus.
           REJECTED);
16     status = decision;
17     // Continues with the execution of next elements in the process
           flow
18   }
19
20   ...
21 }
```

Listing 1: Example of data flow and resource management in the solidity contract of the process Order-to-cash.

contract or the *process contract*. Lines 6-10 define the variables of the process. These variables are defined in the global documentation of the model, from where CATERPILLAR copies them into the smart contract.

Lines 13-18 outline the function `ValidatePO_Complete`, which is generated by CATERPILLAR from the `Validate PO` user task in the running example. Tasks designed to interact with external resources (user and service tasks) may read and write from/to the variables in the process contract. This data mapping is specified in the task's specification using the syntax `<Data_to_export> : <Data_to_import> -> <Operations_to_perform>`. The `<Data_to_export>` section defines which variables are read by the task from the process contract (i.e., input parameters of the task). The `<Data_to_import>` specifies the output parameters of the task, i.e., the data that the task obtains from the external resource. The `Operations_to_perform` section contains a set of Solidity operations to map the output parameters to the variables of the process. For example, the `Validate PO` task takes as input the stock keeping unit (sku), the quantity and the price per unit. It returns the decision that the user makes on the PO (to accept it or reject it). This output parameter is type-checked and then copied to the "status" variable. The corresponding task specification is the following[9]:

[9]The Solidity statement `require` is used as a precondition at the beginning of a function to check if the underlying transaction would fail or not.

```
  (bytes32 sku, uint quantity, uint price) : (POStatus decision)
      ->
   { require(decision == POStatus.ACCEPTED
          || decision == POStatus.REJECTED);
   status = decision; }
```

Listing 2 sketches the contract `OrderToCashWorklist`, which acts as a proxy to handle the interactions generated by user tasks in the ORDER-TO-CASH process. Although any BPMN element that can be triggered by a user generates the corresponding functions in the contract, below, we only discuss the methods of task `Validate PO` to illustrate the approach.

```
1  contract OrderToCashWorklist {
2    struct Workitem {
3      address instanceAddress;
4      uint elementIndex;
5    }
6
7    Workitem[] public workitems;
8
9    event ValidatePO_Requested(uint workitemId, bytes32 sku, uint
         quantity, uint price);
10
11   function ValidatePO_Start(bytes32 sku, uint quantity, uint price)
         external {
12     workitems.push(Workitem(msg.sender, 2));
13     ValidatePO_Requested(workitems.length - 1, sku, quantity, price);
14   }
15
16   function ValidatePO(uint workitemId, uint decision) external {
17     require(workitemId < workitems.length);
18     require(workitems[workitemId].elementIndex == uint(2));
19     require(workitems[workitemId].instanceAddress != address(0));
20     WorklistInterface(workitems[workitemId].instanceAddress).
           ValidatePO_Complete(decision);
21     workitems[workitemId].instanceAddress = address(0);
22   }
23   ...
24 }
25
26 contract WorklistInterface {
27   // == FUNCTIONS IN WORKLIST CONTRACT ==
28   function ValidatePO_Start(bytes32 sku, uint quantity, uint price)
         external;
29   function ValidatePO(uint workitemId, uint decision) external;
30   // == FUNCTION IN MAIN CONTRACT ==
31   function ValidatePO_Complete(POStatus decision) external;
32   ...
33 }
```

Listing 2: Example Worklist and interaction interface of the process Order to cash contract.

For every user task, CATERPILLAR generates two methods in the worklist contract, i.e, `ValidatePO_Start` and `ValidatePO` in `OrderToCashWorklist`. In the process contract, the task is implemented as one function, `ValidatePO_Complete` in Listing 1. To generate these functions, CATER-

PILLAR uses the following approach:

- `<element name>_Start(<Data_to_export>)`. This function must be triggered internally when the corresponding element is reached during the execution flow in the process contract (see control-flow Section 4.3.2 for further details). The method includes as parameters the set of variables annotated as `Data_to_export` in the corresponding BPMN element in the process model. The worklist contract includes the corresponding work-item that is stored in a dynamic array, called `work-items`. A work-item contains the address of the process contract that made the requests and the index that identifies the element in the process contract. Work-items would be accessible by the external resource and would be retrieved from an external application when required. The input parameters and the index of the work-item in the dynamic array that works as the work-item identifier are stored in a Solidity event on the Ethereum log, which is also visible to external applications.
- `<element name>(<Data_to_import>)`. This function should be called by the external resource to provide the information required to continue with the execution of the element. Then, the worklist sends the data to the process contract and marks the work-item as *completed*, by setting the address to 0. The external resource must fulfil any access control policies defined for the process. In the example displayed in Listing 2, any user is allowed to access any function.
- `<element name>_Complete(<Data_to_import>)`. This function, implemented by the process contract, is invoked by the associated worklist. The function first checks that the blockchain address performing the call matches the worklist's address, which is stored as a global variable in the process contract (line 3 in Listing 1). Next, the function calls `<Operations_to_perform>` to update the process variables. The task is then marked as "completed" and the state of the process instance is updated. If a second call arrives from the worklist, it is rejected.

In the presence of duplicate names on elements interacting with external resources, the previous three functions include an extra parameter `uint elementIndex`. Solidity allows the overhead of methods if they have a different definition of parameters. Therefore, CATERPILLAR merges the elements with the same name and same amount and type of parameters in a single function. Next, the three methods are generated with the particularity of executing the actions relying on the value of the parameter `elementIndex`. Note that this parameter is also an attribute of the work-item.

To achieve modularity, CATERPILLAR generates an interface to handle the function calls between the worklist contract and the process-related contracts. Thanks to this interface, we avoid direct interaction between Solidity contracts, which would lead to higher gas con-

sumption during deployment.[10] Instead, we create a third contract (`WorklistInterface` in lines 27-34 of Listing 2), which mediates all calls to the worklist contract. Accordingly, all the function calls are of the form `Contract_Interface(contract_address).function_name(parameters)` (cf. line 21 of Listing 2)).

Service tasks are handled in a similar way as user tasks. The main difference is that for service tasks, the API of the oracle contract is specified in the process model. In contrast, for user tasks, the oracle is implemented by the predefined worklist contract.

4.3.2. Control-flow Perspective

To encode the execution status of each element in a BPMN model, we rely on the following classification of BPMN elements:

External: Elements involving external resources, i.e., user tasks, receive tasks, message catching events and service tasks.

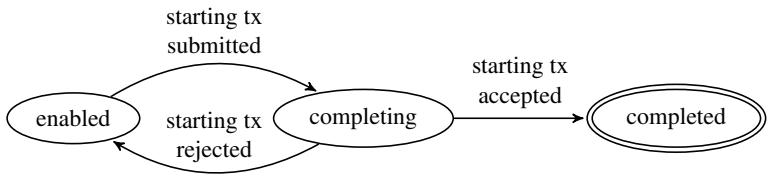
Reusable: Elements whose execution instantiates a process represented by an external process model (not necessarily provided) or which trigger a sub-routine specified within the current process model. This class includes call activities, multi-instance activities, non-interrupting boundary events, and event sub-processes.

Internal: All other elements: script tasks, gateways (exclusive, parallel and event-based), intermediate and end events (default, terminate, message, signal, error, and escalation). We also put embedded (single-instance) sub-processes in this category since they can be in-lined into the parent process and hence do not trigger a separate sub-routine.

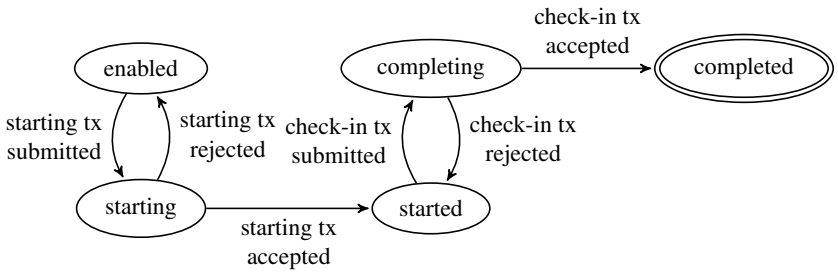
The life-cycle of a BPMN element in CATERPILLAR depends on its class as depicted in Figure 15. Commonly, a BPMN element becomes enabled when a token is present on its incoming edges (one of them in the case of exclusive join gateways, all of them otherwise). Boundary events are enabled if they are attached to a started activity. Finally, an event sub-process becomes enabled if it is included in a process/sub-process which contains, at least, one element enabled or started. The execution of an enabled element consumes the incoming tokens, generating new ones on its outgoing edges as described further below.

The life-cycle of internal elements is the simplest. As there are no external interactions, the element is automatically executed the moment it becomes enabled. This automatic execution happens when a previous element has been executed, and within the scope of the transaction that executed the previous element. For an example, consider the process model in Figure 9 and the contracts in Listing 1 and Listing 2. When a transaction calls the function `ValidatePO`, the transaction executes the respective work-item function, which calls the workflow function

[10] https://medium.com/daox/avoiding-out-of-gas-error-in-large-ethereum-smart-contracts-18961b1fc0c6



(3) Life-cycle of internal elements



(4) Life-cycle of external/reusable elements

Figure 15: Life-cycle of BPMN elements in CATERPILLAR: (4) external/reusable (3) internal

`ValidatePO_complete`; this function, in turn, executes the token flow and calls the function corresponding to the data-based XOR split, cf. Figure 9, and depending on the decision, further functions will be called, all within the scope of the single transaction to `ValidatePO`. In the life-cycle, we refer to this transaction as the "starting tx". The state "completing" is used to capture the validation and inclusion (or not) of transactions on the blockchain (since transactions can be rejected). The state becomes "completed" when this transaction is included in the blockchain. Note that, CATERPILLAR performs every internal operation in a single transaction until reaching a step where only calls to functions in other contracts are pending.

The execution of an external element requires interaction with an oracle, as discussed in Section 4.3.1. When an external element is enabled during the execution from a "starting tx" as above, the oracle contract (e.g., worklist) is invoked, and the element is in state starting/started. Unlike internal elements, an external element requires a separate transaction, called "check-in tx" in the life-cycle, to proceed. Eventually, the oracle makes the corresponding response call, and the element is then marked as "completed".

In the running example, once the task `ValidatePO` is enabled, the contract `OrderToCashProcess` invokes function `ValidatePO_Start` of `OrderToCashWorklist`, which corresponds to the "start" transaction in the life-cycle. Consequently, the task is checked-in when an authorized user triggers the function `ValidatePO` in `OrderToCashWorklist`. Then, the worklist invokes

`ValidatePO_Complete` in `OrderToCashContract`, which in turn generates a new token in the outgoing sequence flow of the task, and the execution continues as outlined above.

Reusable elements also involve interactions with external contracts encapsulating the behaviour of another process. Therefore, the life-cycle is similar to external entities. Once a reusable element is enabled, it creates as many new instances as specified in the corresponding contract. In this case, the element is considered started while any of the instances of the external contracts are running. Intuitively, the element completes when no entity is enabled or started in any of the instances. The simplest case of a reusable element is the call activity. Here, the metadata required to instantiate the contracts of the process associated with the call activity must be provided. Otherwise, an execution error occurs when the call activity becomes enabled. CATERPILLAR creates the contracts of the other reusable elements as described in Section 4.3.3.

The occurrence of an interaction with an external/reusable entity triggers changes in the state of the process instance. Such updates correspond to recomputing the new distribution of tokens over the set of elements in the model and, as a consequence of this, executing internal elements or starting some others that also require interaction with other external entities.

The control-flow perspective of a process is implemented by simulating a token game as specified in the BPMN standard. When a process instance is created, a token is generated by the start event, which traverses sequence flows in the model until reaching the end event(s). To simulate the token game, we assign sequential indexes (starting from 1) to each node in the BPMN model. We use index 0 as the identifier of the process. For the flow arcs, we follow the same approach but starting with zero. Like the work presented in [56], we assume that the model is 1-safe, meaning that it is designed in a way that at most one token is present on a sequence flow at any time. This property allows us to use a bit array, to encode the distribution of tokens in a given state of a process instance. These bit arrays are encoded as 256-bits unsigned integers, which is the default word size in the Ethereum Virtual Machine.

Each process contract generated by CATERPILLAR contains two integer variables called `marking` and `startedActivities` to encode the current state of a process instance. Variable `marking` is a bit-array encoding the distribution of tokens across the sequence flows of the process model. Each sequence flow is associated with one bit in this variable: 1 if the sequence flow has a token, 0 otherwise. Variable `startedActivities` encodes the set of triggered external/reusable elements. Each bit in `startedActivities` corresponds to an external/reusable element. Moreover, a dynamic array named `subInstanceAddresses` stores the addresses of every instance created by each reusable element. Variable `subInstanceStartedIndexes` is a bit array that tracks which of these reusable elements are started. Finally, attribute `worklist` keeps the address of the contract

that handles user tasks.[11]

We use bit-wise operations to handle all the queries/updates on the process state. To check if an element is enabled or started, the bit-wise *AND* allows testing set inclusion. The bit-wise *OR* provides a method to encode the set union as an integer. This operator is used to append tokens in the `marking`, to group the indexes of the incoming edges of an exclusive gateway or the elements contained in a sub-process, among others. Finally, the combination of *NOT* and *AND* serves to remove tokens/elements from the variables `marking`/`startedActivities`.

The control-flow implementation is illustrated in Listing 3, which complements Listing 1 with other attributes and operations required to manage the control-flow perspective. Consider the function `ValidatePO_Complete` shown in lines 14-18. This function is called by the worklist when a user performs the task `ValidatePO`. Line 15 requires `startedActivities & uint(4) != 0`, meaning that the activity corresponding to 4, i.e., `ValidatePO_Complete`, is started. Here, 4 is the decimal representation of the binary number 100, i.e., the bit in the third-last position of the bit array, or index position 2. Note that, as part of a previous transaction, the `if` statement in lines 24-29 must start the interaction with the worklist when the task became enabled (i.e., condition `tmpMarking & uint(2) != 0` is `true`). As result, the token on the incoming arc is removed, `tmpMarking &= uint( 2)`, and the task is started, `tmpStartedActivities |= uint(4)`. The numbers are the indexes assigned to the corresponding arcs and tasks, respectively, when compiling the model into Solidity. Note that we need to cast every bit-mask to avoid overflows in the operations because an integer literal is mapped in Solidity to `uint8` instead of `uint256`.

The computation of the new process state happens inside a function called `step` shown in lines 20-55 of Listing 3. The function `step` is internal, which means that external actors cannot call it. However, when an external entity's function call updates the state of the process contract, the function `step` is invoked.

To illustrate how the `step` function works, consider again function `ValidatePO_Complete` shown in lines 14-18. In line 17, the function `step` is called to update the process state. When making this call, the outgoing edge is activated by changing the `marking`, and `ValidatePO` is marked as completed by changing `startedActivites`. The function `step` receives as input a copy of these changed values of `marking` and `startedActivities`. It is a common practice in Solidity to copy the values of contract variables into local ones, as a way to reduce the number of write operations over contract variables which are costly.[12] Later, function `step` identifies the set of BPMN elements that are en-

[11] In the current implementation, all bit vectors limit the respective content to 256 (e.g., sequence flows). While this limit could easily be lifted, doing so would likely increase the gas cost.

[12] The EVM has three areas to store items. (1) The *storage*, where all the contract state variables are located, and it is costly to use. (2) The *memory* to hold temporary values, which is cheaper. (3) The *stack* to hold small local variables, which is almost free, but it allows a limited amount of values.

```
contract OrderToCashProcess {
  ...
  RuntimeRegistry private registry;

  uint private marking = 1;
  uint private startedActivities = 0;

  address private parent = 0;
  uint private instanceIndex;

  address[] private subInstanceAddresses;
  mapping(uint => uint) private subInstanceStartedIndexes;

  function ValidatePO_Complete(POStatus decision) external {
    require(/* Resource validations */ && startedActivities & uint
        (4) != 0);
    // <Operations_to_perform> --> Data perspective updates
    step(marking | uint(4), startedActivities & uint(~4));
  }

  function step(uint tmpMarking, uint tmpStartedActivities)
      internal {
    while (true) {
      ...
      // User Task (external resource interaction)
      if (tmpMarking & uint(2) != 0) {
        WorklistInterface(worklist).ValidatePO_Start(sku, quantity,
            price);
        tmpMarking &= uint(~2);
        tmpStartedActivities |= uint(4);
        continue;
      }
      // XOR Gateway (internal element)
      if (tmpMarking & uint(4) != 0) {
        tmpMarking &= uint(~4);
        if (poStatus == POStatus.ACCEPTED)
          tmpMarking |= uint(16);
        else
          tmpMarking |= uint(8);
        continue;
      }
      // Call Activity (reusable element)
      if (tmpMarking & uint(16) != 0) {
        tmpMarking &= uint(~16);
        address child = registry.newInstanceFor(uint(3), this);
        uint index = subInstanceAddresses.length;
        subInstanceAddresses.push(child);
        subInstanceStartedIndexes[uint(3)] |= (uint(1) << index);
        AbstractProcess(child).setInstanceIndex(index);
        tmpStartedActivities |= uint(8);
        continue;
      }
      ...
      break;
    }
    marking = tmpMarking;
    startedActivities = tmpStartedActivities;
  }
  ...
}
```

Listing 3: Example of a Solidity contract implementing the control-flow perspective.

abled based on the current marking. Indeed, the function `step` repeatedly executes a sequence of `if` statements to determine whether the current marking enables a given BPMN element. If that is the case, the `step` function starts the execution of the corresponding element. Due to concurrency, more than one element may get enabled in a given state. For this reason, the function `step` has to restart the while loop until no more enabled elements are found. It is only at this moment that the newly computed instance state is stored in the contract variables `marking` and `startedActivities`.

Coming back to the execution of `ValidatePO` in Listing 3, when calling `step` from `ValidatePO_Complete`, the task is completed after updating the corresponding bit from `tmpStartedActivities`, which adds a new token on the arc with index 2. Then, the next exclusive gateway is enabled and executed internally (see lines 31-38). The conditions of an exclusive gateway must be boolean expressions encoded in Solidity; they are attached to the outgoing arcs of the gateway in the model, and embedded in the generated code as shows the line 33.

Finally, we describe how to execute a reusable element. Let us consider that the call activity `GoodsShipment` is enabled, which is handled by the `if` statement in lines 40-49. Checking if the element is enabled and updating the process state is handled analogously to external tasks. However, in this case, a new instance of a related smart contract is created (lines 42-46). The new instance is created through the registry, which requires: (i) the address of the parent, (ii) the index assigned to the child when compiling the model. Accordingly, the registry chooses the factory to create a new instance of the child from the index provided, which requires that a *parent-child-factory* relation exists in the registry and a factory instance is running on the blockchain. Otherwise, it throws an execution error. Then, the parent updates the variables `subInstanceAddresses` and `subInstanceStartedIndexes` with the new child address and status (lines 41-42). Also, the child updates the attributes `parent` and `instanceIndex`. The former one is the address of the parent and is set during deployment through the factory. The latter one keeps the position where the children address was added in the dynamic array of the parent. Finally, the call activity is marked as "started".

4.3.3. Sub-processes and Reusable Elements

CATERPILLAR implements three of five types of sub-processes in the BPMN standard: (1) call activity, which invokes a process defined in a separate process model; (2) embedded sub-process, which invokes a process model embedded inside its parent process; (3) event-sub-process, which is like an embedded sub-process, but that is triggered by an event.

Embedded sub-processes can be inlined in their parent process (and hence are *internal* elements), except when they have a multi-instance marker. In the latter case, CATERPILLAR generates a separate Solidity contract to encode the multi-instance sub-process. This contract will be instantiated once for each instance of

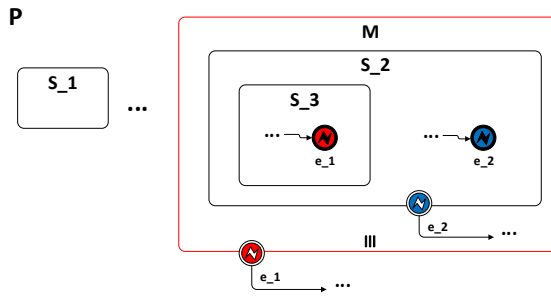


Figure 16: Nested subprocesses with propagation of error events.

the sub-process. In the absence of a multi-instance marker, a nested embedded sub-process is inlined either inside its closest parent (reusable) sub-process or inside the root process. For example, consider the simplified process hierarchy represented in Figure 16.[13] As a result, CATERPILLAR generates two contracts, one for the root process P, including all the atomic elements in P and S_1. The other contract encodes the multi-instance sub-process M. Accordingly, it includes the atomic elements in S_2 and S_3, even when S_3 is not a direct child of M in the hierarchy.

Listing 4 provides a pseudo-code illustrating possible blocks of instructions in the function `step` to handle embedded sub-processes and multi-instance sub-processes(see Figure 16). Lines 6-9 refers how to start the execution of an internal sub-process, i.e, S_1 in P. When the sub-process is enabled in the execution flow, the function `step` automatically executes the start event of the sub-process adding tokens on its outgoing edges. As the elements in S_1 are part of P, the execution continues as described in Section 4.3.2. Note that the functions ending with *mask* refer to the integer encoding arcs/elements into a bit-array.

Although the elements in M are not encoded as functions in the contract of P, the instances of M must be started by P when M is enabled in the execution flow. Listing 4 shows the two possible cases handled by CATERPILLAR. As illustrated in lines 11-19, if M is a parallel multi-instance, the approach differs from instantiating a call activity only in the `for` loop to create several instances. The sub-process remains started in P while at least one instance of M is running (i.e., at least one instance with `marking` or `startedActivities` different from zero). A default event is propagated to P when any instance of M is finished. Thus the marking in P must be updated accordingly. As shown in lines 21-29, sequential multi-instances involve the creation of one instance. Then, the spots required for the next ones are reserved in the dynamic array `subInstanceAddresses`. The next instance is created when the previous one propagates an event notifying its

[13]Note that, the model in Figure 16 is very simplistic outlining only the sub-processes involved in the hierarchy with some error events to handle which are described later in Section 4.3.4, and omitting other possible BPMN elements and sequence flows.

```
1   contract P {
2       ...
3     function step(uint tmpMarking, uint tmpStartedActivities)
          internal {
4       while (true) {
5             ...
6         if (tmpMarking & incomingEdgesMask(S_1) != 0) {
7           tmpMarking & uint(~incomingEdgeMask(S_1)) |
                outgoingEdgeMask(startEvent(S_1));
8           continue;
9         }
10        // Parallel Multi-instance of M
11        if(tmpMarking & incomingEdgesMask(M) != 0) {
12          tmpMarking & uint(~incomingEdgesMask(S_1))
13          for (uint i = 0; i < number_of_instances; i++) {
14            // Create a single instance of the contract M.
15            // Same as call activities, see~\ref{lst:step}, lines
                42-46
16          }
17          tmpStartedActivities |= nodeIndexMask(M)
18          continue;
19        }
20        // Sequential multi-instance of M
21        if(tmpMarking & incomingEdgesMask(M) != 0) {
22          tmpMarking & uint(~incomingEdgesMask(S_1))
23          // Create a single instance of the contract M.
24          // Same as call activities, see~\ref{lst:step}, lines 42-46
25          for (uint i = 0; i < number_of_instances - 1; i++)
26            subInstanceAddresses.push(0);
27          tmpStartedActivities |= nodeIndexMask(M)
28          continue;
29        }
30        ...
31        break;
32      }
33      marking = tmpMarking;
34      startedActivities = tmpStartedActivities;
35      }
36    ...
37  }
```

Listing 4: Subprocess execution through the step function.

ending. The sub-process remains started until the last instance notifies that it finished. For further details, we fully describe the event propagation among contracts in Section 4.3.4.

An event-sub-process is treated as *internal* when triggered by an interrupting event. Otherwise, it would work as *reusable*, and hence a separate contract is generated for it. The rationale behind is that non-interrupting events may be triggered several times during the execution of a process. Thus, they are similar to multi-instance sub-processes. Non-interrupting boundary events are particular cases of reusable elements. Here, CATERPILLAR creates a separate contract including the sub-graph of elements reachable from the event. As a result, we obtain a new node in the process hierarchy with the same behaviour of a non-interrupting event-sub-process.

4.3.4. Event Handling

Event handling in BPMN refers to the actions of throwing and catching events and the state changes arising thereof. Some events change the state of the process trivially: They merely move a token from their input to their output sequence flow(s). However, other events lead to terminate their enclosing process instance and may need to be propagated upwards through the process hierarchy. The latter include end events in event-sub-processes terminate and error events, and boundary events attached to an activity. Below we discuss how CATERPILLAR handles the *event propagation* engendered by such events.

CATERPILLAR distinguishes four cases of event propagation, when an event e is thrown in a sub-process S:

Upward: S propagates e to its parent P in the hierarchy. If P contains a catching event that matches e, then it is handled, and the propagation stops. Otherwise, P propagates e to its parent and so on. If P has no parent (root) and no sub-process could catch e on the path from S to P, the propagation finishes and, depending on the event type, the running instances are stopped or not.

Single upward: S propagates the event to its parent that shall handle it.

Broadcast: S propagates e to the root process. Then, the event flows to any started child sub-process that is reachable by traversing the hierarchy from the root. Accordingly, any enabled catching event with the same type of e must handle it when reaching the corresponding sub-process.

Outside: S pushes the result of the event to a resource interacting with the process from outside of the hierarchy.

The propagation always starts when a throwing event is enabled. These type of events are internal, thus executed in the function `step`. If the propagation does not traverse any reusable sub-processes, the token distribution resulting from the propagation is computed at compilation-time and encoded in the `step` function. If, on the other hand, the propagation traverses reusable elements, the resulting token distribution cannot be computed at compilation-time. The pseudo-code in Listing 5 illustrates how the propagation of throwing events is handled in the `step` function. The pseudo-code includes, as comments, the operations/analysis to perform when compiling the model, as well as the generated Solidity code. For conciseness, parameters `tmpMarking` and `tmpStartedArtivities` are shortened as `tM` and `tS`, respectively. The instructions to check if an event e is enabled and for removing tokens from its incoming edges is common to all cases (cf. lines 4-5). The rest of the generated Solidity code is case-dependent.

The handling of error or escalation events follows the upward pattern, referred to as Case 1 in Listing 5. At compilation time, we search for the corresponding catching event. This search is local to a reusable sub-process, i.e. only embedded sub-processes implemented in the contract of the reusable sub-process are traversed. If required, i.e. unsuccessful local search, the event is propagated to the parent contract by calling function `propagateEvent` as shown in lines 7-9.

```
1       function step(uint tM, uint tA) internal {
2         while(true) {
3           ...
4           if (tM & /*incomingEdgesMask(e)*/ != 0) {
5             tM &= uint(~/*incomingEdgesMask(e)*/);
6   /* (1) Case: Upward --> Error or Escalation
        ---------------------------------------- */
7   /*       e' = closestLocalCatchingEvent(e) */
8   /* (1.1) IF NOT exist e' */
9             (tM, tA) = propagateEvent(/*type(e)*/, /*code(e)*/, /*index
                  (e)*/, tM, tA);
10  /* (1.2) ELSE IF isInterrupting(e') AND isStartEvent(e')
11            C = subprocessEnclosing(e') */
12            (tM, tA) = killProcess(/*index(parent(C))*/, tM, tS);
13            tM |= /* initialMarkingMask(C) */
14  /* (1.3) ELSE IF isInterrupting(e') AND isBoundaryEvent(e')
15            C = subprocessWhereAttached(e') */
16            (tM, tA) = killProcess(/*index(C)*/, tM, tS);
17            tM |= /*outgoingEdgesMask(e')*/;
18  /* (1.4) ELSE IF isNotInterrupting(e') */
19            createNewSubprocessInstance(/*index(e')*/);
20            tS |= /*index(e')*/;
21  /* (2) Case: Single upward --> Default End Event
        ---------------------------------- */
22  /*      S = subprocessEnclosing(e)     */
23            if((tM & /*fullEdgesMask(S)*/) | (tA & /*fullNodesMask(S)*/
                  ) != 0)
24  /* (2.1) IF S IS Internal */
25              tM |= /*outgoingEdgesMask(S)*/;
26  /* (2.2) ELSE */
27              (tM, tA) = propagateEvent("Default", /*code(e)*/, /*index
                    (e)*/, tM, tA);
28  /* (3) Case: Single upward --> Terminate, given S =
        subprocessEnclosing(e) ---------- */
29   /* IF S IS Reusable FROM Non-Interrupting Boundary Event  */
30            (tM, tA) = propagateEvent("Terminate", /*code(e)*/, /*index
                  (e)*/, tM, tA);
31   /* ELSE IF S IS Reusable */
32            (tM, tA) = killProcess(0, tM, tA);
33            (tM, tA) = propagateEvent("Default", /*code(e)*/, /*index(e
                  )*/, tM, tA);
34   /* ELSE */
35            (tM, tA) = killProcess(/*index(S)*/, tM, tA);
36            tM |= /*outgoingEdgesMask(S)*/;
37  /*   (4) Case: Broadcast --> Signal
        ---------------------------------------------------- */
38            (tM, tA) = propagateEvent(/*type(e)*/, /*code(e)*/, /*index
                  (e)*/, tM, tA);
39          }
40          ...
```

Listing 5: Event propagation.

The effect of this function depends on the type of event. For example, a *terminate* event finishes the current process instance. An *error* event may be caught by the process scope under which it occurs (which may lead to terminating the current instance), but it may also need to be propagated upwards to the parent process instance. Meanwhile, a signal event is broadcast down to any reachable

children. In order to handle uncaught events, every process contract implements a function `handleEvent`, which allows a sub-process instance to indicate that an uncaught event has been generated under their scope.

Each process contract implements a function `killProcess`, which terminates an instance of a subprocess given its index. In other words, it removes tokens on the arcs enclosed in the input subprocess and terminates any enabled child recursively. Generating the function `killProcess` in a contract involves roughly the following steps. (i) Find the activities that, in addition to the root process (with index 0), can be interrupted in the contract, e.g., those with an interrupting boundary event attached. (ii) For any possible activity to terminate, generate a block of instructions that validates if the index of the input subprocess matches to update the token marking accordingly. (iii) If the activity is reusable, then update `startedActivities` and recursively terminate the related active instances, and later update the global field `subInstanceStartedIndexes`.

Continuing with the upward propagation, lines 10-20 in Listing 5 show how to handle an event caught internally. If the catching event is interrupting, two cases are distinguished: either the event starts an event-sub-process, or it is attached to the boundary of an activity. The former one terminates the current instance of the sub-process enclosing the event-sub-process, which is later restarted and enabled by adding a token in the outgoing arc of its start event. The second case terminates the sub-process where it is attached to the boundary event, which redirects the execution flow through its outgoing edge. Catching a non-interrupting event creates a new instance of the contract generated from that event in the same way described in Listing 3 lines 42-46. In such a case, any related sub-process continues the execution unaltered. Besides, several catching events can be triggered as a result of one propagation. The standard keeps open the behaviour to follow if no sub-process catches a propagated event. Hence, CATERPILLAR interrupts any involved sub-processes in case of error propagation. Otherwise, the execution stays unchanged.

Cases 2 and 3 in Listing 5 illustrate the *single upward* propagation. It occurs when a sub-process finishes by reaching a *default end event* or a *terminate* event. A *default* event is thrown if no token exists in the `marking` and no child sub-process is active, as checked in line 23. Here, `fullEdgesMask` and `fullNodesMask` represent the bit-masks of every edge/node in the sub-process containing the event *e* to throw. Two cases are distinguished here: (2.1) *e* is enclosed in an internal sub-process, i.e., embedded and (2.2) *e* is enclosed in the reusable sub-process that defines the current a contract from which the event will be propagated to another contract. Case 3 handles the *terminate* event, which terminates the sub-process instance in which it occurs and, in the case of a reusable sub-process, it propagates a "Default" event to its parent.

Case 4 in Listing 5 deals with signal events. This case involves calling function `propagateEvent` repeatedly until reaching the root process. Once in the root, the event is disseminated to all subprocesses in the hierarchy by calling function

`broadcastSignal`. Every contract implements this function, which triggers any catching signal events and propagates the call to any reusable child recursively. Unlike upward propagation which comes from a child, catching/propagating a signal requires checking if the corresponding element is enabled. Note that boundary events and event-sub-processes do not fit in the typical token game because they have no incoming edges. Therefore, checking if they are enabled relies on verifying the element to/in which they are attached/included depending on the case. Handling the signal when it is caught follows the same approach described above for error/escalation, depending on whether the event is interrupting or not.

To illustrate the approach, consider the example in Figure 16. Error event *e_2* enclosed in sub-process *S_2* is thrown in the `step` function of the contract generated from *M*. This event is caught at the boundary of *S_2*, which is embedded in *M*. Thus, handled by the `step` function which already started the propagation as outlined in Case 1.3 of Listing 5. Consequently, the sub-processes *S_2* and *S_3* are terminated in the current instance of *M*, and the execution continues through the outgoing sequence flow of the boundary event with the same label (*e_2*). On the other hand, the propagation of *e_1* starts in *M* but is handled by the contract defined from *P*. To this end, the `step` function calls `propagateEvent` in *M*, which terminates the current instance and propagates the event by invoking function `handleEvent` defined in *P*, which handles the event at the boundary of *M*. As a result, the remaining running instances of *M* are terminated, and the execution continues via the outgoing flow of the boundary event.

A smart contract generated from a non-root process always calls the function `handleEvent` to propagate an event to its parent. This function checks the data received about the event (e.g., type, code, which child/instance is the source) to handle it accordingly. Indeed, the approach is similar to the one described in Listing 5, i.e., trying to catch the event locally, if not possible, then propagating it to the parent. However, a particular scenario must be solved when receiving a *default* event from a contract encoding a multi-instance sub-process. Here, dealing with a sequential multi-instance involves checking if any reserved spot is empty (see Listing 4) to create a new instance. When no running instance exists, in both the parallel and sequential case, the execution continues by calling the function `step` to add a token on the outgoing edge of the sub-process. For example, in Figure 16, if *P* receives a default event from *M*, then the source instance is marked as finished. If there are no other running instances of *M*, then the function `step` in *P* is executed to add a token in the outgoing edge of *M* (not included in Figure 16) and continuing the execution.

4.4. Implementation and Evaluation

The Off-chain Runtime of CATERPILLAR and the web-based user interface are implemented in Node.js[14] and rely on the standard Solidity compiler solc-js[15] to compile the smart contracts. Interactions with running instances of the smart contracts are supported via the Ethereum client geth.[16] The functionality of the CATERPILLAR's Off-chain Runtime is exposed via a REST API described in the following. Subsequently we describe the evaluation, which is focused on feasibility, correctness, and cost of our approach and implementation.

4.4.1. REST API

CATERPILLAR's REST API is built around four types of resources summarized in Table 1: (i) *models*, including BPMN models, the associated compilation artefacts and their instantiation, (ii) *processes*, which refers to process instances deployed, running or completed on the blockchain, and (iii- iv) *worklists* and *services* which involves interactions of user and external services with the running instances of a process. The component named "Deployment Mediator" in the architecture in Figure 10 exposes its functionalities through the resource *models*, while the "Execution Monitor" responds to the REST actions involving the resources *processes*, *worklists* and *services*. The component "BPMN Compiler" involves no REST interaction, but is invoked from the "Deployment Mediator" when a request to register a new model is received. Note that CATERPILLAR assumes the models created by the "Modeling Panel" are aimed to be deployed into the blockchain. Thus, the "Deployment Mediator" triggers the compilation of such models before deploying them.

As making changes in the process at runtime is not recommended in compiled approaches, we are not exposing such kind of functionalities in the compilation-based engine. Therefore, the compiled version of CATERPILLAR requires that the addresses where the running smart contracts linked to call activities and service tasks must be annotated in the corresponding process model. However, extending the REST API and user interface to allow updates in the "Runtime Registry" regarding the links/relations is straightforward given that the on-chain functionalities are implemented in the "Runtime Registry". In this direction, the "Deployment Mediator" will also serve as the entry point to register and link process contracts, factories, worklists, and services, that are not necessarily produced by CATERPILLAR but relying on the structure outlined by its interfaces. Besides, the smart contracts encoding worklists and factories are generated with a default policy which allows any external resource to instantiate a process and execute a

[14] https://nodejs.org/en/

[15] https://github.com/ethereum/solc-js

[16] https://github.com/ethereum/go-ethereum/wiki/geth

Table 1: CATERPILLAR's compilation-based engine REST API.

Verb	URI	Description
POST	/models	Registers a BPMN model (triggers also code generation, compilation and updates the registry with a default configuration for the given model, i.e., after this operation the process model is ready to be instantiated, unless the operator incurred errors)
GET	/models	Retrieves the list of registered BPMN models
GET	/models/:m-hash	Retrieves a BPMN model and its compilation artefacts
POST	/models/:m-hash	Creates a new process instance from a given model
GET	/models/:m-hash/instances	Retrieves all the instances created from a given process model
GET	/processes/:p-address	Retrieves the current state of a process instance
PUT	/worklists/:wl-address/workitems/:wi-index	Checks-in a work item (i.e. user task)
PUT	/services/:s-address/tasks/:t-index	Executes a service task

started task.[17]

The compilation-based version of CATERPILLAR requires that the addresses where the running smart contracts linked to call activities and service tasks must be annotated in the corresponding process model. The smart contracts encoding worklists and factories are generated with a default policy which allows any external resource to instantiate a process and execute a started task. However, extending the REST API and user interface to allow updates in the "Runtime Registry" regarding the links/relations is straightforward given that the on-chain functionalities were implemented already. In this direction, the "Deployment Mediator" will also serve as the entry point to register and link process contracts, factories, worklists, and services, that are not necessarily produced by CATERPILLAR but relying on the structure outlined by its interfaces.

A user can submit a BPMN model to be compiled and deployed using an HTTP POST request on the URL `/models` which is also the approach followed by CATERPILLAR's "Modeling Panel". The request is made with a JSON message which includes the model serialized in the BPMN 2.0 XML standard format. Querying the process model metadata from the "Distributed Repository" uses HTTP GET requests, as shown in Table 1. An HTTP GET request on the URL `/models` returns the information of all the models stored in CATERPILLAR's repository. Since this data can be voluminous, CATERPILLAR yields only a list containing the name and hash reference for each model stored in the repository. Then a user can retrieve all the information associated with a particular model (i.e., serialized BPMN model, Solidity code, etcetera) using an HTTP GET with the model's corresponding hash.

Given the identifier of a process, i.e., hash produced when stored in the repository, a user can use an HTTP POST to request the creation of an instance. After submitting the transactions to create and start the instance, CATERPILLAR re-

[17] For a further description of what kind of access control mechanism implements CATERPILLAR, we refer the reader to the Chapter 6 of this thesis.

```
{
    "process-identifier": "o2c-hash",
    "href": "/processes/o2c-address",
    "workitems": [
      {
        "elementId": "Request_Quote_Id",
        "name": "Request_Quote",
        "importParameters": [
          { "type": "uint", "name": "quote" }
        ],
        "instances": [
            {
              "exportParameters": [],
              "href": "/worklists/wl_address/workitems/wi_1",
            }
        ]
      },
      {
        "elementId": "Submit_Quote_Id",
        "name": "Submit_Quote",
        "importParameters": [],
        "instances": [
          {
            "exportParameters": ["type": "uint", "name": "quote", "
                value": "100"],
            "href": "/worklists/wl_address/workitems/wi_3"
          }
        ]
      }
    ]
    "services": []
}
```

Listing 6: Sample process instance's state (model from Figure 9).

trieves the URL where the status of the newly created instance can be accessed. In the implementation of the compilation-based engine, the URL associated with a process instance includes the address of the underlying contract. Moreover, once the corresponding transactions are included in the blockchain, the "Runtime Registry" publishes a Solidity event with the address of the new instance in the Ethereum Log. Thus, the "Event Monitor" accordingly sends notifications with this address to the "Execution Monitor" and "Execution Panel".

At any time, a user can query the state of a process instance by using an HTTP GET request on the URL associated with the instance of interest. In response, a JSON message is sent, with the information required to visualize and execute any started user/service task. By way of example, consider the ORDER TO CASH process model presented in Figure 9 and assume the execution of a process instance has progressed up to the point where two instances of the sub-process CARRIER SELECTION have started (Figure 9(b)). Moreover, consider that a participant executed the task `Request quote` in one instance of the sub-process CARRIER SELECTION. Listing 6 shows the JSON message sent responding to the HTTP GET request on the URL `/processes/o2c-address`, where the acronym "o2c" references in a compact way the root process ORDER TO CASH.

First, the JSON message in Listing 6 provides the hash identifier and the URL to access the process instance. Next, two lists named workitems and services contain the information of started user and service tasks, respectively. For each element, its identifier and name in the BPMN model are retrieved. For the workitems, the import parameters are listed, which must be provided by a participant when executing the task. Note that the type and name of such parameters are used by the "Execution Panel" to generate web forms that the user must fill. Finally, there is a list with the URLs to execute the task through an HTTP PUT and the parameters to export, whose values must be displayed when visualizing the task. Note that the export parameters are derived from information stored in the smart contract, which varies among instances. For example, the quote provided by a shipper when executing the task `Submit quote` – which also starts the task `Request quote` – is sent to the carrier who can decide whether to submit such a quote or not. The addresses where the instances of the sub-processes GOODS SHIPMENT and CARRIER SELECTION are running are not needed (and not returned), because CATERPILLAR forces user tasks to be executed through the corresponding worklist.

As indicated above, the execution of a user task requires an HTTP PUT to a URL provided to that end when querying the process state, e.g., `/worklists/wl_address/workitems/wi-_1` in line 14 of Listing 6. Such URL contains as parameters the address when running the worklist and the index of the corresponding work-item. Furthermore, the request needs to include a JSON message with the expected values for the task.

4.4.2. Experimental Setup

Our experimental evaluation aims at assessing the cost of executing business processes using CATERPILLAR, relative to other baselines that either only record the execution of the process (without enforcing it) or that do not fulfil the design principles outlined in the introduction of this thesis(cf. Section 1.2).

In line with existing works on blockchain-based collaborative process execution [56], we used four datasets for the evaluation, each of them consisting of a BPMN process model and corresponding event log. Table 2 presents the statistics of the datasets. The first dataset referred to as *Invoicing*, is an event log of a real-world business process, used and distributed by Minit[18] for demonstrating its process mining tool. The BPMN model for this dataset was derived from the event log, using a state-of-the-art process discovery tool [7]. Since the BPMN model is automatically discovered, some traces ($< 1\%$) were non-conforming. Thus, we discarded them since they represent a limited subset of all traces.

The other three datasets, i.e., *supply chain*, *incident management* and *insurance claim*, are process models extracted from the literature, and were used in the experiments reported in [163]. In this case, the event logs for the experiments

[18] http://www.minitlabs.com/ - last accessed 17/05/2018

Table 2: Datasets used in the evaluation.

Process	Tasks	Gateways	Trace Type	Traces
Invoicing	40	18	Conforming	5,317
Supply chain	10	2	Conforming	5
			Not conforming	57
Incident Management	9	6	Conforming	4
			Not conforming	120
Insurance claim	13	8	Conforming	17
			Not conforming	262

were generated after the BPMN models using a simulation tool. In order to assess CATERPILLAR's ability to differentiate between conforming and non-conforming inputs, we inserted noise into these event logs with non-conforming traces, by randomly changing events over some of the traces also selected at random.

We replayed the distinct log traces and interacting with CATERPILLAR using its REST API. To that end, we implemented a replayer component, which compiles, configures and deploys each one of the four process models in the input datasets. After compiling and deploying the process models, the replayer reads the corresponding event log and sequentially feeds each the events in the log into the CATERPILLAR system through its REST API. In order to execute a task, the replayer queries the state of the process instance, formats a JSON message with the required data, and submits the request to CATERPILLAR. Then, the replayer waits for a notification from CATERPILLAR's "Event Monitor", indicating that the block containing the underlying transaction has been completed, and additional information that includes the transaction hash and gas consumption.

The replayer queries CATERPILLAR that in turn checks the on-chain runtime to determine if there is a work-item matching the next event in the event log. If the response to the replayer indicates that there is no such work-item (meaning that the on-chain runtime rejects the event in question), the replayer marks the trace as non-conforming, skips the event, and continues reading the next event in the log.

For comparison, we run the same experiments using three baselines. The first baseline, namely *Basic* is designed to reflect the approach where the process is executed in one or more off-chain BPMSs, and the blockchain is only used to leave tamper-proof execution traces of the process via a connector between the BPMS and a blockchain platform [6, 119]. In this baseline, the smart contract of a process instance records a reference to each event executed in this process instance, using a dynamic array of `bytes32`. Here, we assume the data generated by each event is stored off-chain. The second baseline, namely *Default* corresponds to a smart contract that enforces the control-flow of the process and stores the data required to evaluate the conditions in the decision gateways of the BPMN model, but without any work-item management (i.e., no handling of user and service tasks) and with all runtime components left off-chain. In other words, this approach does not

satisfy design principle # 4 in Section 1.2 – the smart contracts generated from the process model rely on other off-chain runtime components. This approach is used in [56]. The third baseline, namely *Optimized*, is the same as *Default* with some code optimizations to reduce the number of bits required to store the state of process instances, as outlined in [56].

The experiments were run on a personal computer with an Intel i5-5200 dual-core CPU. Moreover, we run them over *testrpc*,[19] which is a NodeJS-based implementation of Ethereum client used for development purposes. In this way, both CATERPILLAR's runtime and log replayer ran over the same computer.

4.4.3. Experimental Results and Discussion

Given that gas consumption is deterministic, the traces were grouped together such that only one execution was performed for each distinct trace in the event log. The same approach was taken in an existing work [56]. As expected, all non-conforming behaviour was handled correctly by CATERPILLAR: the request corresponding to a non-conforming event was ignored by CATERPILLAR because the underlying task was not enabled (life-cycle state: "executing").

The measurements of gas consumption are shown in Table 3. The table reports the cost of instantiation of the process (in gas) and the runtime cost, which means the cost of handling all the events related to a given process instance. The costs reported in this table are adjusted such that they reflect the average overall gas consumption. The last two columns of the table report the relative overhead of CATERPILLAR regarding each of the baselines. For example, an overhead of 3.51 for the instantiation cost implies that the compilation-based engine of CATERPILLAR consumes 3.51 times more gas than the baseline in question.

As expected, Table 3 shows that the cost of instantiation of the *Basic* approach is considerably lower than all other approaches. The latest is because the smart contract generated by this approach is relatively small: it the contract exposes a function that takes an event as input and records it on-chain. On the other hand, the execution costs of this contract are comparable to that of CATERPILLAR since the *Basic* contract has to make a write operation on a dynamic array, for each event. In contrast, the *Default* and *Optimized* approaches do not store each event, but instead, they store the state of the process instance in a bit-set, which is a less costly operation. CATERPILLAR does the same, but it also performs other operations in addition to updating the state of the process instance.

Table 3 also shows that, on average, the smart contracts generated by CATERPILLAR consume two to three times more gas than that required by the solidity code generated by *Default* and *Optimized*. This trend was expected, as the code generated by *Default* and *Optimized* is encapsulated in a single smart contract and deals only with basic control-flow and data-flow perspectives. In contrast, due to more advanced architectural design, CATERPILLAR produces several smart con-

[19] https://github.com/0xProject/testrpc

Table 3: CATERPILLAR's compilation-based engine: process instantiation and execution costs.

Process	Tested Traces	Translator	W. Avg. Cost		Relative Overhead	
			Instant.	Exec.	Instant.	Exec.
Invoicing	5316	Basic	123,625	589,228	22.9	1.8
		Default	1,089,000	383,109	2.60	2.84
		Optimized	807,123	297,351	3.51	3.66
		CATERPILLAR	**2,830,063**	**1,088,315**	–	–
Supply chain	62	Basic	123,625	570,444	8.9	0.99
		Default	304,084	281,206	3.62	2.02
		Optimized	298,564	272,186	3.69	2.08
		CATERPILLAR	**1,100,590**	**566,861**	–	–
Incident mgmt.	124	Basic	123,625	375,929	9.0	0.86
		Default	365,207	185,680	3.07	1.75
		Optimized	345,743	166,345	3.24	1.95
		CATERPILLAR	**1,119,803**	**324,420**	–	–
Insurance claim	279	Basic	123,625	1,008,840	10.8	1.23
		Default	439,143	552,274	3.05	2.24
		Optimized	391,510	514,712	3.42	2.40
		CATERPILLAR	**1,338,152**	**1,235,617**	–	–

tracts to support concerns such as work-item handling and runtime housekeeping among other things. One single interaction between CATERPILLAR's off-chain and on-chain components may involve chains of interactions between up to four solidity smart contracts, e.g., *workflow*, *worklist*, *factory* and *registry*.

Therefore, it is worth noting that the comparison between the costs reported for CATERPILLAR and those reported for *Default* and *Optimized* is not straightforward, because the functionality provided by CATERPILLAR is more sophisticated than that provided by the other two options. Moreover, the costs associated with CATERPILLAR's code can still be reduced by applying the techniques described in [56], which are those that are associated with *Optimized*. The latter is left as a venue for future research.

Another drawback to consider in blockchain-based systems is the latency, i.e., the time required for a transaction to be appended into the blockchain. We did not perform any experiment to measure the latency as it is an issue related more to the blockchain platform than the CATERPILLAR system itself. For example, at the moment of writing this thesis, the average time between blocks in the public Ethereum blockchain is 13.35s, however in January 2020, that value was around 17.0s, and in October 2015 it reached a peak of around 30.0s. [20] Other parameters, like the gas price in Ethereum, impact the time a transaction is mined. Two studies [163, 175] which estimated the latency on blockchain-based systems, also considered private Ethereum networks in which they restricted the complexity, so the inter-block times ranged from 2.3s and 6.3s. The studies showed that latency

[20] https://etherscan.io/chart/blocktime

is low in private and customized blockchains. However, on the public blockchain, it may be high for scenarios like fast trading, and acceptable in supply chain scenarios in which seconds of latency is not an issue [163]. Accordingly, as shown in the works [163, 175], the processes executed by CATERPILLAR may face latency issues depending on the blockchain platform selected.

4.5. Summary

This chapter addressed our first research question, *How can the high-level abstractions of BPMSs be combined with the capabilities of blockchain technology to support the execution of collaborative business processes between mutually untrusted parties?* To that end, we presented the design and implementation of the CATERPILLAR system for blockchain-based execution of collaborative business processes captured in the BPMN notation. Specifically, this chapter focused on a compiled approach which exploits the immutability of the generated smart contracts as a source of trust, thus once deployed, the smart contracts cannot tamper. These compiled approaches prevent changes in the process at runtime. Thus they are suitable for processes where the tasks and their relationships would remain unchanged during the entire execution.

Furthermore in this chapter, we discussed the architectural components of the compilation-based engine of CATERPILLAR, including either on-chain and off-chain components, that supports an approach of compliance-by-design, i.e., the execution controlled by transactions which are accepted if and only they comply with the collaborative process model. Specifically, the execution engine relies on a novel BPMN-to-Solidity compiler which translates BPMN models into smart contracts in the Solidity language. The BPMN-to-Solidity compiler supports hierarchical processes models, which includes complex constructions like sub-processes, multi-instance activities, event handling and specialized tasks. The engine supports process models enhanced with data constraints. Besides, it also provides a generic worklist handler to allocate tasks to the process participants, among other components.

Finally, we performed an empirical evaluation to asses the cost of executing business processes using CATERPILLAR. As expected, the experimental evaluation shows that CATERPILLAR can handle realistic process models. However, it also suggests that the approach would not scale to extensive process models with hundreds or thousands of elements. However, this scalability limitation is inherent to public blockchains, and not to the CATERPILLAR system itself. Thus it can be mitigated by the migration of the system to consortium blockchain technologies, such as Hyperledger.

5. INTERPRETED EXECUTION OF BLOCKCHAIN-BASED BUSINESS PROCESS MODELS

In Chapter 4 we introduced the CATERPILLAR system with the focus on an execution engine that follows a compiled approach. These compiled approaches certainly address the problem of lack of trust by generating smart contracts that are immutable but keeping aside other essential properties like flexibility and efficiency. To fill this gap, in this chapter, we focus on the research question **RQ2**: *How can collaborative processes involving mutually untrusted parties be flexibly and cost-efficiently executed on a blockchain platform?* As a solution, we propose a BPMN interpreter, which is also integrated into the CATERPILLAR system. As disused in Section 1.1.2, interpreted approaches fit better for processes which are subject to changes at runtime, or to be executed on public blockchains in which the participants must pay a transaction fee for the deployment.

The chapter is structured as follows. First, Section 5.1 describes the new architecture of the system to support an interpreted execution. Next, Section 5.2 presents the dynamic data structures to store the control-flow and data perspectives. Then, Section 5.3 delves into the inner workings of the interpreter. Finally, Section 5.4 discusses the implementation and evaluation, while Section 5.5 summarises the key aspects of our proposal.

5.1. Extending the Architecture of the Caterpillar System with the Interpretation-based Engine

The proposed blockchain-based interpreted execution engine follows the same design principles and principles as the compilation-based version of CATERPILLAR(cf. Section 1.2). Specifically, the system is designed to enable a set of untrusting parties to develop, deploy and execute collaborative processes on blockchain in a tamper-proof manner. To that end, the full state of the process execution, as well as the process execution logic itself, are recorded on the blockchain, so that no party is able to execute a transaction that does not abide to the agreed-upon process model.

To illustrate our proposal, we use the BPMN model in Figure 17, where a number identifies each element. The model contains two user tasks (T1 and T2) to be performed by process participants, and which also serve to check-out/in process data. The remaining elements require no interactions with external actors (i.e., they are performed internally). Event E1 triggers the instantiation of the process, while events E2 and E3 end the execution. Gateway G1 checks conditions, based on the process data, to split the flow into two exclusive paths (joined later via G2). Script task T3 updates the process data by executing internal scripts. Call-activities S1 and S2 reference two sub-processes which are modelled sepa-

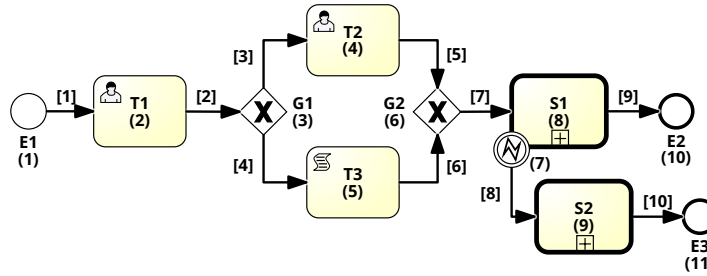


Figure 17: Simple BPMN model.

rately. An error event may interrupt sub-process S1. This error event is caught by the boundary event attached to S1 and re-directs the flow of control into the exception flow leading to S2.

Figure 18 illustrates the architecture of the system that is organized into three layers. In the bottom, the "On-Chain and Storage Layer" encloses the set of smart contracts that control the process execution, namely "On-chain Components", which are replicated across all the nodes of a blockchain network, e.g., the public Ethereum network. Besides, only the data relevant to the process execution is stored on-chain, while compilation/parsing artefacts are stored off-chain in the "Process Repository". In the middle, the "Off-chain Access Layer" includes a set of tools to parse, compile, deploy, execute and monitor business processes in the "On-Chain and Storage Layer". Finally, the "Process-Aware Layer" comprises a set of components to model/execute the process guided by high-level and model-driven interfaces. Note that the components in the "Off-chain Access Layer" and "Process-Aware Layer" run outside of the blockchain. Thus they can be tampered. However, the state of each process instances is stored on-chain, and all the decision points are evaluated on-chain. Besides, each actor in a collaborative process can host the off-chain components separately. Thus, each actor can check directly from the blockchain which actions were performed by others.

From a high-level perspective, several components in the "Process-Aware Layer" and "Off-Chain Access Layer" in the interpretation-based engine exposes similar functionalities as the "Web portal" and the "Off-Chain Runtime" in the compilation-based engine. However, the architectures of both engines are different, i.e., they share no component. We avoid mixing the components because both engines follow different approaches, so the participants may decide which one is more convenient based on the specificities of the process they want to execute.

Compared to the WfMC and BPMS-RA reference architectures [66, 124], the component in the architecture of the CATERPILLAR interpretation-based engine can be classified as follows. (i) *Process Definition* comprehends the "Modeling Panel" and "Process Repository", as they serve to define and to store all the information regarding the high-level process models. (ii) *Workflow Enactment Service* encloses the "BPMN Compiler", "BPMN Parser", "Registration Mediator", "Deployment Mediator", "Runtime Handler", "BPMN Interpreter", "Control Flow",

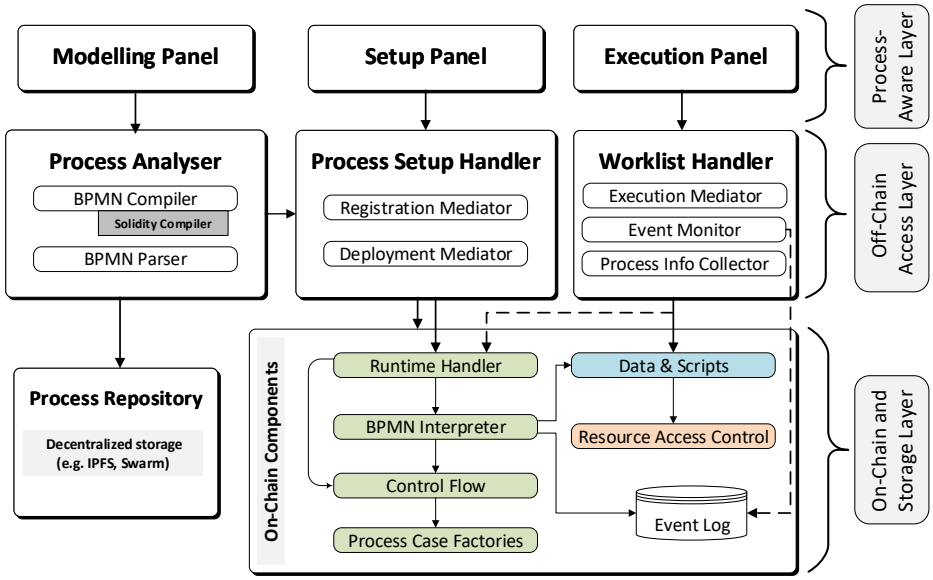


Figure 18: Extended architecture of the CATERPILLAR system: the interpretation-based engine.

"Process Case Factories" and "Data & Scripts". They are responsible for the parsing, creation, management and execution of the workflow instances. (iii) *Administration and Monitoring Tools* relates to the "Execution Mediator" and 'Process Info Collector" and "Resource Access Control". They handle resource and user access control management as well as the execution of enabled tasks. (iv) *Workflow Interoperability* includes "Event Monitor" and "Ethereum Log", as they support the interactions between the on-chain components with external systems. Specifically, to request an external interaction and event is placed in the "Ethereum Log", and later processed by the "Event Monitor" (off-chain). (v) *Workflow Client Functions* comprises the "Modeling Panel", "Setup Panel" and "Execution Panel", which enable the interaction of the end-users with the CATERPILLAR's off-chain and on-chain components. Below, we provide a more detailed description of the components in each one of these layers.

5.1.1. On-Chain and Storage Layer

The "On-Chain Components" are the core that handles the process execution. In the sake of reusability, the process perspectives, i.e., control-flow, data management, and resource allocation, are decoupled into different components. The smart contracts in the components on the left (i.e., "Runtime Handler", "BPMN Interpreter", "Control Flow" and "Process Case Factories") implement the set of general operations to any process model. Thus, they are hard-coded only once based on the BPMN standard and the system requirements. In contrast, the smart contracts in the component named "Data & Scripts" contain the model-specific data and operations that need to be extracted from each process model. The com-

ponent "Resource Access Control" handles the user access control and resource allocation. Finally, the event log is provided by some blockchains, like Ethereum, giving to off-chain components convenient access to events generated on-chain.

The component "Control Flow" stores the information about the structure of the process models, their elements and relations. Given that a process model may include sub-processes, the data structure is a tree where each node, named IFLOW, represents a sub-process that keeps references to its children (if exist). Besides, the nodes map for each enclosed BPMN element the model-related information to be used by the "BPMN Interpreter" to handle the execution. For example, some of the information to store can be the type of element (task, event, gateway, etcetera), the incoming/outgoing arcs, to what sub-process an event is attached, and so on. Accordingly, each node needs to be deployed once per sub-process in the model, and the corresponding blockchain address identifies it. The address of the root node would identify the full process model. Unlike compiled approaches, the control-flow perspective is not statically encoded as a smart contract. Instead, the information is collected off-chain from the model and added to the corresponding nodes dynamically.

The "Process Case Factory" includes the set of contracts to instantiate and start the execution of a process. Thus, when a sub-process is linked as a child in the IFLOW hierarchy, the parent has to store the address of the corresponding factory to instantiate the sub-process during the execution. In the following, we will refer to process instances as process cases, to differentiate them from smart contract instances.

As the name suggests, the smart contracts in the component "Data & Scripts" implement the data perspective. Data requirements are process-dependent, strongly typed, and their values are conditioned and scoped by the process cases, e.g., a variable defined in a sub-process may store different values in each sub-process case. Accordingly, smart contracts must be compiled from the process model. The scripts related to user/service/script tasks and the conditions to decide the paths in exclusive gateways mostly interact with the process data. Thus, such instructions are also compiled from the model into the contract implementing the data perspective. The interactions of external actors via user/service tasks also requires two operations per task: check-out to request data from the process case, and check-in to send data to the process and to proceed with the execution. The smart contracts in this component form a hierarchy with a node per (sub-)process case which we will refer to as IDATA. Each IDATA node stores the sub-process state and keeps a reference to the related IFLOW node. Indeed, the factories linked to a sub-process, in the IFLOW hierarchy, must define how to instantiate the smart contract of the corresponding IDATA node.

Figure 19 illustrates some of the relations among the smart contracts deployed to execute two cases of the process modelled in Figure 17. In Figure 19, each node represents one smart contract instance, while the background colours differentiate the four smart contracts involved. The control-flow perspective requires a single

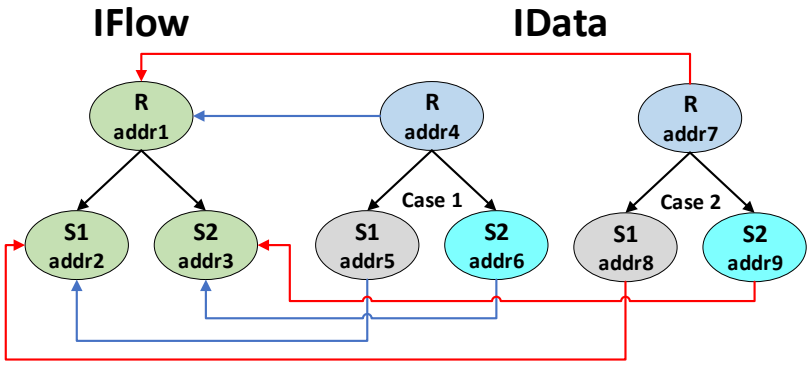


Figure 19: Graphical representation of the the control-flow and data perspectives smart contracts deployed to execute two cases of the process in Figure 17.

IFLOW data structure, that is instantiated once per sub-process in the model. On the other hand, three different smarts contracts (one per sub-process) are required on the data perspective to encode general operations of the IDATA structure plus the data/scripts compiled from each sub-process. Each IDATA node keeps a reference to the corresponding IFLOW node that is used by the "BPMN Interpreter" to check the control-flow information of the process case, e.g., to update, after executing an element, to verify which others can be executed. Although not in the figure, only one factory is required per sub-process, e.g., both instances `addr4` and `addr7` of `R` are started by the same factory.

The "Data & Scripts" component serves as the entry point for external actors to access data and execute tasks. Nevertheless, the access would be restricted by a set of smart contracts in the component "Resource Access Control" derived from binding policies as presented in Chapter 6.

The "BPMN Interpreter" is a single smart contract that implements the process execution logic defined by the BPMN standard. This component keeps no information about any of the process perspectives, but queries/updates such data from/into the IFLOW and IDATA structures.

The "Runtime Handler" keeps tracks of the process instances, binding policies and their relation with other smart contracts. The "Event Log" provides a source for communication between off-chain and on-chain components. Out of the "On-Chain Components", the "Process Repository" stores and provides access to compilation artefacts and metadata to link the Solidity code to elements of the BPMN models. The operation of the "Runtime Handler", "Event Log" and the "Process Repository" is similar to the equivalent components in the architecture of the compilation-based version of CATERPILLAR (cf. Chapter 4, Section 4.2 for a complete description).

5.1.2. Off-Chain Access and Process-Aware Layers

The "Off-Chain Access Layer", in the middle of Figure 18, provides a service-oriented entry point for external applications to interact with the "On-Chain and

Storage Layer".

The "Process Analyser" (on the left) extends the CATERPILLAR's "BPMN Compiler" to generate the IDATA structure from a BPMN model. The "BPMN Compiler" uses a standard SOLIDITY COMPILER to produce the metadata and interfaces that are used to deploy and execute the smart contracts. Additionally, the "BPMN Parser" extracts the control-flow information from the model that is structured to be inserted in the IFLOW hierarchy (see Section 5.2).

The "Process Setup Handler" (in the middle) serves as the entry point to deploy smart contracts (e.g., produced by the "BPMN Compiler"), and to update the IFLOW structure. The "Deployment Mediator" provides the set of operations to deploy the IFLOW and IDATA hierarchies, as well as the factories and the resource access control contracts. On the other hand, the "Registration Mediator" supports the operations to update the IFLOW structure (e.g., insert BPMN elements into nodes). Besides, the "Registration Mediator" allows to change the relations among the smart contracts in the "On-Chain Components", e.g., to link/unlink sub-process as nodes into IFLOW, update the access control policies, etcetera.

On the right of the "Off-Chain Access Layer", the "Worklist Handler" enables external actors to query the process state and data, as well as to execute tasks on a given process case. The compilation-based engine of CATERPILLAR implemented the "Worklist Handler" as smart contracts, i.e., on-chain. Besides, the worklist allowed only interaction of human actors, while non-human actors (e.g., information systems, IoT devices) are handled via another on-chain component named "Service Bridge". In the current approach, we restrict the resource allocation and access control by policies that support dynamic bindings of actors to roles (see Chapter 6) implemented by the "Resource Access Control" component. The latter removes the need to use static worklist contracts generated per process model to validate any data checked-in into the process instances. Thus, the "Worklist handler" is implemented off-chain what is less costly. Besides, the binding policies would verify blockchain accounts that are controlled indistinctly by users, groups, systems, or (IoT) devices. Thus the "Service Bridge" is joined into the "Worklist Handler" off-chain. Accordingly, an actor to check-in a task via the "Execution Mediator" provides the process case (i.e., address of the corresponding IDATA node) and the identifier of the task. Then, the "Execution Mediator" interacts with the "Data & Scripts" component, that in turn verifies the actor rights via the "Resource Access Control" component. If the actor can perform the task, the process data and state is updated. For that, the corresponding IDATA node invokes the "BPMN Interpreter" that in turn interacts with the related IFLOW nodes. The "Process Info Collector" interacts with the "Runtime handler" and "Data & Scripts" to query the active process cases, control-flow addresses, the state of a given process case, and to check-out data from user/service tasks. Finally, the "Event Monitor" listens for events generated from the smart contracts, e.g., to notify that the state in some process case was updated.

On top of the architecture, the "Process-Aware Layer" exposes the function-

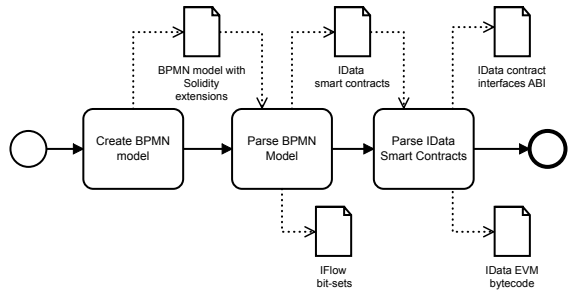


Figure 20: Parsing of BPMN models on the CATERPILLAR's interpretation-based engine

ality of the "Off-Chain Access Layer" to end users (e.g., process administrators and workers) via a form-based user interface. The rationale of this layer is the same as the "Web Portal" described in Chapter 4, Section 4.2. The "Modelling Panel" allows the user to draw the BPMN models. The "Setup Panel" supports the updates of the data structures of different perspectives and their relations. Finally, the "Execution Panel" interacts with the "Worklist Handler" to retrieve all the information about deployed models, running instances, and to allow executing tasks by stakeholders.

Figure 20 illustrates how the interpretation-based engine parses a BPMN model. First, a process participants should agree in the process model. To that end, they can use the "Modelling Panel" or any other modelling tool. Then, the "Process Analyser" produces the set of smart contracts related to the IDATA structure (including the factories to create process instances), and transforms the process control-flow into bit-sets as required by the IFLOW data structure. In a second step, CATERPILLAR passes the smart contracts generated from IDATA to the "Solidity Compiler" which produces the EVM bytecode and ABI, required to deploy the smart contracts in the blockchain. Note that, all the metadata generated from transforming the model is stored in the process repository, so all the participants may check them at any time.

Figure 21 shows how to perform the set-up of the process after completing the parsing of the model. Furthermore, it illustrates how to create process instances. Note that, all the tasks starting with the verb "Instantiate" take as input the ABI and EVM bytecode from the corresponding smart contracts, and retrieving the address where the smart contract instance is running. In a first step, one instance of the "Runtime Handler" and the "BPMN Interpreter" must be created respectively. These contracts are independent of any model so that they can be instantiated at any time before the process execution (through the "Deployment Mediator") and then reused.

Before creating the first process instance, some components need to be setted-up. First, a participant should create an instance of the data structure IFLOW (initially empty), and later updated with the control-flow related bit-sets produced

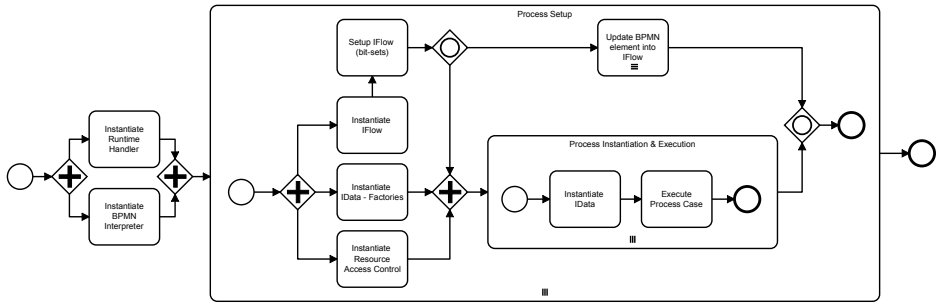


Figure 21: Process set-up and instantiation on the CATERPILLAR's interpretation-based engine.

from parsing the model. Also, a participant should instantiate the factory contract, used to create the process instances by instantiating the IDATA structure. Similarly, an instance of the contract "Resource Access Control" is required to handle the dynamic binding of actors into roles (see Chapter 6). As for the compilation-based engine, the "Runtime Handler" tracks the relations between the on-chain components, i.e., factories, control-flow, resources, data, once a participant sets-up a process though the "Deployment Mediator" (for creating new instances) and the "Registration Mediator" (to update the data structures).

On the creation of a process instance, the participant acting as case-creator interacts with the "Deployment Mediator", which in turns invokes the corresponding factory contracts. During the execution, the "Resource Access Control" allows participants to nominate or release actors dynamically (enforced by policies), and accordingly restricts the execution of the tasks. On the other hand, the IFLOW data structure provides the control-flow information to update the process state after executing a task. Off-chain, the components in the "Worklist Handler" monitor the process execution by listening if new events are written in the "Event Log", and recovering the process state, i.e., which tasks are enabled to be executed from the corresponding work-item.

Note that, in Figure 21 the `Process Setup` and `Process Instantiation & Execution` are represented as parallel multi-instance sub-processes, as a way to illustrate how the components can be reused. Also, the task sequential multi-instance `Update BPMN element into IFlow` shows how the IFLOW structure can be updated at any time, even after the instantiation of several process cases. The later, as discussed in Section 1.3 may lead to inconsistencies, especially if multiple instances points the same IFLOW structure. Therefore, CATERPILLAR allows such updates if all the participants of the corresponding cases agree on it.

The flexibility mechanisms allowed through the task `Update BPMN element into IFlow` deserves further explanations. More specifically, the interpretation-based engine supports, the flexibility requirements variability, adaptability, evolution and looseness [128]. For example, participants may handle different variants to the same process represented as separate instances of the data structure

IFLOW (i.e., support for variability requirement). Note that the three main process perspectives, i.e., control-flow, data and resources, are decoupled, then the participants can agree on how to couple them for each process case. Similarly, the support for the adaptability and evolution requirements on the control-flow relies on the fact that participants can temporarily or permanently deviate the process execution by modifying the IFLOW data structure, e.g., by adding or removing tasks. However, as discussed in Section 1.3, those modifications may lead to deadlocks. Because of that, each participant must privately validate and agree on the updates. In the case of looseness, the compilation-based and interpretation-based engine support late-binding and late modelling of sub-processes, restricted by agreement policies (see Chapter 6). Therefore, a venue of future work is to extend the agreement policies to support also the mechanisms related to the variability, adaptability and evolution requirements automatically.

5.2. Control-Flow and Data Representation

As hinted in Section 5.1, the IFLOW structure is the tree that captures the hierarchical representation of a process model with each node enclosing the control-flow information of the corresponding sub-process. During the parsing of the model, each BPMN element is associated with an integer index that is unique per sub-process. Similarly, arcs are enumerated as well. The mapping element-index can be accessed from the "Process Repository" that also provides tamper-proof storage. Figure 17 shows a possible numeration for both arcs and elements in the represented process model. Such indexes serve later to encode the elements into bit-sets to implement the operations effieciently using bit-wise operators.

An IFLOW node can be updated via three operations. First, the operation `setElement` updates or inserts an element depending on whether it is already contained. The operation requires as input the element index `eInd`, the incoming `preC` and outgoing `postC` arcs, and the element description `typeInfo`. The `preC` and `postC` are bit-sets with 1s in the bits corresponding to the indexes of the arcs contained in the set, and 0s on the remaining. The element description `typeInfo` is also encoded as a bit-set such that each characteristic is identified by a bit (see Figure 22). The second operation `linkSubprocess` add a child into a IFLOW node. Here, the index of the sub-process/call-activity in the parent must be provided and the address when running the child IFLOW node. Besides, the number of sub-process instances to create, and the list (can be empty) with the indexes of the attached events are required. Note that the indexes provided must correspond to elements already added in the parent node; otherwise, the operation will be rejected. For example, to link a sub-process to the call-activity labelled as S1 in Figure 17, we have to provide the call-activity index, i.e., 8, the blockchain address of the IFLOW node created for S1, the index of the attached error event, i.e., 7, and 1 as the number of instances to create. As the factories are related to the IDATA smart contracts, they are updated separately, but before the corresponding

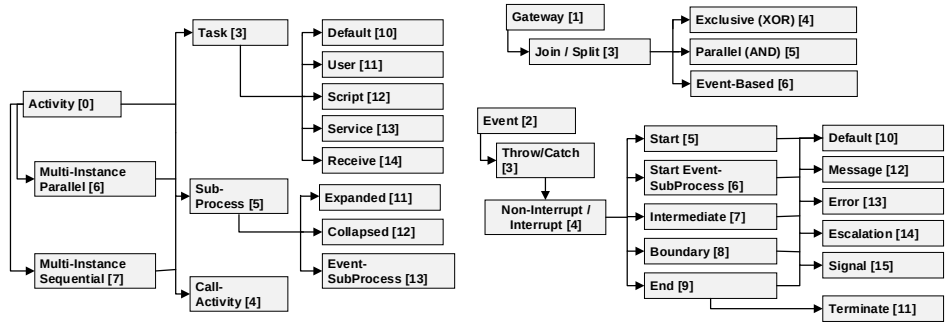


Figure 22: Bit associated to each element/characteristic when encoding the element description as `typeInfo`.

element is reached during the execution of a process case. The operations to query the IFLOW structure are straightforward, thus omitted in this document.[1]

Figure 22 shows how to encode the element description as `typeInfo` from the bits associated (in brackets) with the elements supported by the interpreter. For example, user and service tasks are identified by the bits 11 and 13 respectively, but as they are also activities and tasks, then they must share those bits (0 and 3) too. Besides, to verify if an element is a user task, the bits 0 and 11 must be checked because the terminate event is also identified by the bit 11, but the bit 2 points it as an event. Some bits can encode two characteristics. For example, in the gateways the value 1 in the third bit represents a *join*; otherwise, it is a *split*. Similarly, bits 3 and 4 of an event identify whether it is throwing/catching and interrupting/non-interrupting, respectively.

Listing 7 illustrates a sample of the IDATA smart contract generated from the root process in Figure 17. To generate the IDATA structure we use the same annotations in the BPMN models defined in Chapter 4, Section 4.3. First, the process variables in lines 3-4 are copied from the global documentation of the model. When compiling a sub-process, the variables are extracted from the documentation of the corresponding element. A single function `execScript` manages the execution of scripts. It takes as input an element index, executes the scripts associated to the corresponding element, and returns the bit-set with the outgoing arcs to proceed with the process execution, or zero if the element index is not found. Lines 6-15 shows the body of the `execScript` that encodes the exclusive gateway G1 and the script task T3. The outgoing arcs of the exclusive and inclusive gateways contain boolean expressions encoded in Solidity which verifies the process variables. Then, based on the evaluation of the expression, the execution should be redirected to the corresponding outgoing arc, (cf. lines 7-9). Similarly, the documentation of script tasks includes Solidity instructions to update the process data (cf. lines 10-13).

The functions `checkIn`, and `chekOut` are generated from the documenta-

[1]The full definition and implementation of the IFLOW and IDATA can be accessed from `http://git.io/caterpillar`.

```
1  contract ProcessIData is IData {
2     // == PROCESS VARIABLES ==
3     bool t1Field;
4     bool t2Field;
5     // == SCRIPTS TO EXECUTE ==
6     function execScript(uint eInd) public returns(uint) {
7          if(eInd == 3) { // Gateway G1
8              if(t1Field) return 8; // 1 << 3
9              else return 16;       // 1 << 4
10         } else if(eInd == 5) {    // Script Task T3
11             // Execute script defined by the task
12             return 64;            // 1 << 6
13         }
14         return 0;
15    }
16    // == CHECK IN/OUT FUNCTIONS ==
17    function checkIn(uint eInd, bool _input1) public {
18         if(eInd == 2)           // User Task T1
19             t1Field = _input1;
20         else if(eInd == 4)      // User Task T2
21             t2Field = _input1;
22         revert("Not Found");
23    }
24    function checkOut(uint eInd) public view returns(bool) {
25          if(eInd == 4)          // User Task T2
26              return t1Field;
27          revert("Not Found");
28     }
29 }
```

Listing 7: Example of the Root IData node produced from model in Figure 17.

tion of the tasks supporting external interactions in the model. Accordingly, each task should be annotated with expressions of the form $(Data\_to\_Export) : (Data\_to\_Import) \rightarrow \{Operations\_to\_Perform\}$, to restrict what data must be read/written from/to the process and the operations to perform when executing the task. Intuitively, *Data_to_Export* is returned by the corresponding `chekOut` function, and *Data_to_Import* serves as input in the `checkIn` function that also execute *Operations_to_Perform*. Besides, elements with the same combination of parameter types are grouped into the same function. For example, consider the user tasks T1 and T2 in Figure 17 are annotated respectively with:

$$() : (\textbf{bool}\ \_t1Field) \rightarrow \{t1Field = \_t1Field;\},$$

$$(\textbf{bool}\ t1Field) : (\textbf{bool}\ \_t2Field) \rightarrow \{t2Field = \_t2Field;\}.$$

As both tasks import a boolean variable, they are encoded in the same `checkIn` function as shown in lines 17-23 of the Listing 7. The generation of the `chekOut` function (cf. lines 24-28) only includes T2 because T1 contains no data to export. Listing 7 only illustrates the compiled part of the IDATA nodes. Common operations to query/update the IDATA structure are straightforward, thus omitted.

Each sub-process in the model will produce a IDATA smart contract which is instantiated via a factory that is mapped to the corresponding sub-process in the

IFLOW structure. The hierarchical relationships among the IDATA nodes are built internally during the execution when the corresponding sub-processes are reached in the control-flow. Indeed, the IDATA contracts will produce several hierarchy instances, one per process case. Besides, external actors are only allowed to create instances of IDATA contracts related to the root process.

5.3. BPMN Interpreter Operation

The "BPMN Interpreter" uses six operations to execute process cases based on the BPMN standard. In the following, the notation IDATA/IFLOW(address) refers to a node in the corresponding hierarchy (i.e., a smart contract instance) identified by its blockchain address (i.e., variables ending with `Addr`). Besides, the variable `pState` represents the process state, i.e., two bit-sets comprising the token distribution on edges and the indexes of sub-processes under execution, respectively. For the sake of clarity, the bitwise operations are replaced by functions with names remarked in bold. For example, the functions with suffix `Tokens` would update a bit-set representing the edges containing tokens. The keyword `this` is used to invoke the functions implemented by the interpreter. Besides, the types of BPMN elements are written with capital letters.

Pseudocode in Listing 8 illustrates the function `executeElements`. It receives as input the blockchain address `iDataAddr` of an IDATA node, and the index `eInd` of the element to be executed. Due to security requirements, external actors do not interact directly with the "BPMN Interpreter". Instead, an actor checks-in tasks via an IDATA node, that in turn (after verifying the actor privileges) calls `executeElements` to proceed with the process execution. Indeed, lines 2-3 in Listing 8 would reject any call from addresses distinct to the input IDATA node or the interpreter itself referred as `this`. Elements like gateways, script tasks, throwing events, etcetera, which not require interaction with external resources, are executed internally.

Before executing an element, the interpreter requests the related IFLOW node and the process state from the input IDATA node (see lines 4-5 in Listing 8). The candidate elements, starting by the input `eInd`, are added into a queue in the same order they are reached in the control-flow. The execution follows a Breadth-First Search on the process model until no candidates are available in the queue. In each iteration, the element on the top of the queue is extracted and processed based on its control-flow information. First, lines 10-11 check if the element is enabled, i.e., based on `typeInfo` and checking whether the required tokens are placed on the incoming arcs to enable the element. Such verification uses bitwise operations on the bit-sets `postC` and `pState`. If the element is enabled, the tokens on the incoming arcs are removed, i.e., it is not enabled anymore, and the element is executed based on its `typeInfo` (lines 13-36).

Multi-instance activities are split into two cases, both invokes a function `createInstance` implemented by the interpreter (Listing 9) to create the new

```
1  function executeElements(iDataAddr, eInd) public
2    if (msg.sender != (iDataAddr or this))
3      throw 'REJECTED'
4    iFlowAddr = IData(iDataAddr).getIFlowNode();
5    pState = IData(iDataAddr).getSubProcessState();
6    queue = new Queue(eInd);
7    while(!queue.isEmpty())
8      eInd = queue.pop();
9      (preC, postC, typeInfo) = IFlow(iFlowAddr).find(eInd);
10     if (!isEnabled(preC, typeInfo, pState))
11         continue;
12     removeTokens(pState, preC);
13     switch (typeInfo)
14       case PARALLEL_MULTI_INST:
15         for(i = 1 to IFlow(iFlowAddr).getCountInst(eInd))
16           this.createInst(eInd, iDataAddr);
17         addSubProcess(pState, eInd);
18       case SEQ_MULT_INST || SUB_PROCESS || CALL_ACTIVITY:
19         this.createInst(eInd, iDataAddr);
20         addSubProcess(pState, eInd);
21       case SCRIPT_TASK || EXCLUSIVE_GATEWAY_SPLIT:
22         postC = IData(iDataAddr).execScript(eInd);
23         addTokens(pState, postC);
24       case TASKS and GATEWAYS /* Remaining */:
25         addTokens(pState, postC);
26       case THROW_EVENT:
27         IData(iDataAddr).updateProcessState(pState);
28         evtCode = IFlow(iFlowAddr).getEventCode(eInd);
29         this.throwEvent(iDataAddr, evtCode, typeInfo);
30         pState = IData(iDataAddr).getSubProcessState();
31         if(isCompleted(pState))
32           return;
33         if(INTERMEDIATE_EVENT in typeInfo)
34           addTokens(pState, postC);
35       default:
36         continue;
37     foreach (outEInd in IData(iDataAddr).outElements(eInd))
38       outInfo = IFlow(iFlowAddr).getTypeInfo(outEInd);
39       if(!(EXTERNAL_ELEMENT_INTERACTION in outInfo))
40         queue.push(outEInd);
41   IData(iDataAddr).updateProcessState(pState);
```

Listing 8: Pseudocode of `executeElements` in "BPMN Interpreter".

IDATA nodes and to update the `pState` with the index of the corresponding subprocess (lines 14-20 in Listing 8). Parallel multi-instances produces as many nodes as specified by the model. Sequential multi-instances generate only the first node because the other nodes require the completion of the process case represented by the previous node, which involves the catching of an end event. Thus, they are instantiated by another function called `tryCatchEvent` (See Listing 11).

Script tasks and exclusive split gateways in lines 21-23 of Listing 8 execute scripts compiled from the process model into the IDATA node, which also returns the outgoing arc to update the process state. The remaining tasks and gateways, not involving scripts, are executed by adding the tokens in `postC` to the process state, as shown in lines 24-25. Note that tasks checked-in by external actors may

```
1  function createInstance(iDataAddr, eInd)
2    iFlowAddr = IData(iDataAddr).getIFlowNode();
3    chIFlowAddr = IFlow(iFlowAddr).getChildIFlow(eInd);
4    factoryAddr = IFlow(chIFlowAddr).getFactory();
5    if(factoryAddr == address(0))
6      throw 'REJECTED'
7    chIDataAddr = IFactory(factoryAddr).newInstance();
8    IData(chIDataAddr).setParent(iDataAddr, chIFlowAddr);
9    IData(iDataAddr).addChild(eInd, chIDataAddr);
10   eInd = IFlow(chIFlowAddr).getInitElement();
11   this.executeElements(chIDataAddr, eInd);
```

Listing 9: Pseudocode of `createInstance` in "BPMN Interpreter".

also include scripts, but they are executed by the corresponding check-in function in the IDATA node. Accordingly, only the external task received as input in the `executeElements` is added into the queue, i.e., they are never added/executed internally by the interpreter during the execution (see lines 39-40).

Continuing with Listing 8, lines 26-34 handle the throwing of events by calling the function `throwEvent`. The propagation of the event across the IDATA hierarchy may provoke the interruption of the sub-process represented by the corresponding node, e.g., as a result of handling an error event. Thus, the execution continues only if the process state contains some element enabled after the propagation (lines 31-32). In the case of intermediate throwing events, their outgoing arcs must be added to the process state (lines 33-34). Finally, the loop in lines 37-40 adds each adjacent element (reached via an outgoing arc) as a candidate into the execution queue.

Listing 9 illustrates the sequence of steps required to create a node in the IDATA hierarchy in a given process case. The input is the IDATA node to be the parent and the index associated the child sub-process in the IFLOW structure. First, the factory mapped to the corresponding sub-process is requested, throwing an error if no factory exists (lines 3-6). Next, the new child IDATA node is created via the factory, and the relation parent-child is updated (lines 7-9). Finally, the function `executeElements` is performed in the new child, to ensure that only elements that require external interaction remain enabled. A particular case occurs when an external actor instantiates a root node. There, the root IFLOW node must be provided, and no relation parent-child is added.

Listing 10 illustrates some validations to perform before propagating a thrown event to the parent. The input is the IDATA where the event is thrown, the event code (required for it to be caught) and its `typeInfo`. First, if the event is a message, a blockchain event will be written in the event log to notify that a certain point of the process execution was reached (lines 3-4). Default and Message end events are propagated to the parent only if the execution in the current node (sub-process) is finished (lines 5-7). Remaining events, i.e., error, escalation, terminate, always propagate to the parent. The terminate event must stop the current node before the propagation performed by the function `killSubProcess`.

```
1  function throwEvent(iDataAddr, evtCode, typeInfo)
2    pState = IData(iDataAddr).getSubProcessState();
3    if(MESSAGE in typeInfo)
4      emit MessageSent(evtCode);
5    if(DEFAULT or MESSAGE in typeInfo)
6      if(isCompleted(pState))
7        this.tryCatchEvent(iDataAddr, evtCode, typeInfo);
8    else
9      if(TERMINATE in typeInfo))
10        this.killSubProcess(iDataAddr);
11     this.tryCatchEvent(iDataAddr, evtCode, typeInfo);
```

Listing 10: Pseudocode of `throwEvent` in "BPMN Interpreter".

Listing 11 describes how an event thrown from a node is handled in the parent. The input is the IDATA node where the event is thrown, the event code and its `typeInfo`. The propagation is stopped in lines 3-6 if no parent exists, also finishing the process execution if the received event is an error. Lines 7-10 queries the information about parent and child stored into variables with prefixes `catch` and `subP` respectively. Then, if the execution in the child is completed, the parent updates the state by removing the sub-process and adding a token on its outgoing arc (lines 11-17). In the case of sequential multi-instance activities, lines 18-19, if any instance is pending to be created, then the next IDATA node is created (cf. note that function `executeElements` creates the first node).

Events like terminate, message and default propagate to the parent to notify that a child is finished. Thus, they are not caught by another event as for signals, errors, and escalations. Lines 21-27 in Listing 11 shows how signal events propagate to the root and later broadcast to each running sub-process. Errors and escalations are handled by the parent if exist a catching event matching the code of the event propagated (lines 28-29). If the event is caught, two cases may occur. First, in lines 32-39, if the event is starting an event-sub-process, then the parent is killed if the catching event is interrupting, and a new instance of the event-sub-process is created as the only child. Otherwise, the event-sub-process runs in parallel with the enabled elements in the parent. The second case occurs if the event is caught in the boundary of the sub-process that is throwing it (lines 40-49). Then, the sub-process is ended if the catching event is marked as interrupting. Also, a token is added on the outgoing arc of the boundary event, and the execution proceeds by calling the function `executeElements`. Finally, in line 50, the event is propagated if it cannot be caught in the current node.

```
1  function tryCatchEvent(iDataAddr, evtCode, typeInfo)
2    catchIDataAddr = IData(iDataAddr).getParent();
3    if(catchIDataAddr == address(0))
4      if(ERROR in typeInfo)
5        this.killSubProcess(iDataAddr);
6      return;
7    catchIFlowAddr = IData(catchIDataAddr).getIFlowNode();
8    pState = IData(iDataAddr).getSubProcessState();
9    subPInd = IData(iDataAddr).getIndexInParent();
10   subPInfo =  IFlow(catchIFlowAddr).getTypeInfo(subPInd);
11   if(isCompleted(IData(iDataAddr).getSubProcessState()))
12     IData(catchIDataAddr).decreaseInstCount(subPInd);
13   subPCount = IData(catchIDataAddr).getCountInst(subPInd);
14   if(subPCount == 0)
15     removeSubProcess(catchIDataAddr, subPInd);
16     postC = IFlow(catchIFlowAddr).getPostC(subPInd);
17     addTokens(catchIDataAddr, postC);
18   else if(SEQ_MULTI_INST in subPInfo)
19       this.createInstance(catchIDataAddr, subPInd);
20   if(!(MESSAGE or DEFAULT or TERMINATE) in typeInfo)
21     if(SIGNAL in typeInfo)
22       while(catchIDataAddr != address(0))
23         iDataAddr = catchIDataAddr;
24         catchIDataAddr = IData(iDataAddr).getParent();
25       IData(iDataAddr).updateProcessState(pState);
26       this.broadcastSignal(iDataAddr);
27       return;
28     foreach(ev in IFlow(catchIFlowAddr).getEventList())
29       if(IFlow(catchIFlowAddr).getEvtCode(ev) == evtCode)
30         evInfo = IFlow(catchIFlowAddr).getTypeInfo(ev);
31         attchTo = IFlow(catchIFlowAddr).getAttachedTo(ev);
32         if(EVENT_SUB_PROCESS_START in evInfo)
33           if(INTERRUPTING in evInfo)
34             this.killSubProcess(catchIDataAddr);
35             pState = EMPTY;
36           this.createInstance(catchIDataAddr, attchTo);
37           addSubProcess(pState, attchTo);
38           IData(catchIDataAddr).updateProcessState(pState);
39           return;
40         else if(BOUNDARY in evInfo && attchTo == subPInd)
41           if(INTERRUPTING in evInfo)
42             this.killSubProcess(iDataAddr);
43             removeSubProcess(pState, subPInd);
44           postC = IFlow(catchIFlowAddr).getPostC(ev);
45           addTokens(pState, posC);
46           IData(catchIDataAddr).updateProcessState(pState);
47           next = IFlow(catchIFlowAddr).getOutElement(ev);
48           this.executeElements(catchIDataAddr, next);
49           return;
50     this.throwEvent(catchIDataAddr, evtCode, typeInfo)
```

Listing 11: Pseudocode of `tryCatchEvent` in "BPMN Interpreter".

The remaining two functions, `killElements` and `broadcastSignal`, traverse each descendant reachable from a source node. The function `killElement` takes a IDATA node as input and updates the process state in such node as empty (all bits are set to 0), repeating recursively the same procedure for each child that remains running. The function `broadastSignal` performs a strategy similar as

in lines 28-49 of Listing 11, but catching only signal events, for each running sub-process from the root IDATA node. As the only difference, signals do not require to match the `evtCode`, i.e., each catching signal attached/contained to/in an enabled sub-process must be handled.

5.4. Implementation and Evaluation

The "Off-Chain Access Layer" and "Process-Aware Layer" are implemented in Node.js.[2] The smart contracts are compiled using the standard Solidity compiler solc-js.[3] The deployment and interaction with running instances of the smart contracts are supported via the Ethereum client Geth.[4] The full implementation of the system proposed in this paper can be downloaded under the BSD 3-clause "New" or "Revised" License from the Caterpillar's repository at `https://github.com/orlenyslp/Caterpillar`, version V3.0.

The functionality of the "Process-Aware Layer" is exposed via the REST API described in Table 4. The REST API is built around three types of resources: (i) `interpreter` which manages the deployment of the "BPMN Interpreter" and the operations derived from the parsing of the models, (ii) `i-flow` which involves the deployment and interactions with IFLOW nodes, e.g., to update BPMN elements, link sub-processes and factories, create new process instances, etcetera and (iii) `i-data` which refers to the interactions with IDATA nodes, e.g., to verify the process state, and check-in/out tasks. The full documentation of the REST API, including the format of the messages used in the requests/responses of each operation, can be found in the CATERPILLAR's repository.

In the following, we describe an experimental evaluation aimed at assessing the costs of executing business processes using the interpreted approach presented in this paper, relative to existing compiled solutions on blockchain-based process execution [56, 90, 163]. Accordingly, we used the same four datasets, consisting of a BPMN model and the corresponding event log. The first dataset, named *Invoicing*, corresponds to a real-world business process, used and distributed by Minit.[5] The other datasets referred to as *Supply chain*, *Incident mgmt.* and *Insurance claim* were extracted from the literature and used on the experiments reported in [163]. These datasets were used on the evaluation of the CATERPILLAR's compilation-based engine, thus described in Chapter 4, Section 4.4.2.

Like in the compilation-based engine of CATERPILLAR (cf. Chapter 4), we implemented a component which replays the distinct log traces interacting with the REST API described in Table 1. The replayer parses each of the four BPMN models in the datasets, deploys the contracts of the "BPMN Interpreter" and

[2] https://nodejs.org/en/

[3] https://github.com/ethereum/solc-js

[4] https://github.com/ethereum/go-ethereum/wiki/geth

[5] http://www.minitlabs.com/

Table 4: CATERPILLAR's interpretation-based engine REST API.

Verb	URI	Description
POST	/interpreter	Creates a new instance of the "BPMN Interpreter"
POST	/interpreter/models	Parses a BPMN model. This operation may update the required ON-CHAIN COMPONENTS and PROCESS REPOSITORY (if specified), thus the process would be ready to be executed.
GET	/interpreter/models/	Retrieves the list of parsed BPMN models
GET	/interpreter/models/:m-hash	Retrieves a BPMN model, its compilation artefacts and IFLOW root node instances
POST	/i-flow	Creates a empty IFLOW node
PATCH	/i-flow/element/:cf-address	Updates a BPMN element into a given IFLOW node
PATCH	/i-flow/child/:cf-address	Links a child node (i.e., associated to a sub-process) in a given IFLOW node
PATCH	/i-flow/factory/:cf-address	Relates a factory with a sub-process in a given IFLOW node (i.e. a related IDATA smart contract must exist)
GET	/i-flow/:cf-address/	Retrieves the information (i.e., elements, child sub-process and factories addresses) from a given IFLOW node
POST	/i-flow/p-cases/:cf-address	Creates a new process case from a given IFLOW root node.
GET	/i-flow/p-cases/:cf-address	Retrieves all the process cases created (i.e. IDATA instances) from a given IFLOW root node.
GET	/i-data/:pc-address	Retrieves the current state of a given process case
GET	/i-data/:pc-address/i-flow/:e-index	Checks-out a task in a given process case
PATCH	/i-data/:pc-address/i-flow/:e-index	Checks-in a task in a given process case

Table 5: CATERPILLAR's interpretation-based engine: setting-up costs.

Process	BPMN Elements	Avg. Reg. Cost.
Invoicing	60	110,760
Supply chain	15	105,516
Incident mgmt.	18	114,671
Insurance claim	24	112,850

IFLOW nodes, and updates each IFLOW node with the corresponding BPMN elements and factories. Once the configuration of the models is completed, the replayer reads the corresponding log, and sequentially instantiates each process case (IDATA node) and executes the corresponding events in the log via the REST API. Besides, the replayer collects and assesses the gas consumed by each operation once the corresponding transaction is included in the blockchain. The experiments were performed on a Node.js based Ethereum client named ganache-cli,[6] which simulates a full client for developing and testing purposes on Ethereum.

Table 5 presents the costs in gas derived from setting-up the IFLOW structure at runtime (not required by the compiled approaches). The column labelled as *Avg. Reg. Cost* shows the average costs of registering a BPMN element into the corresponding IFLOW node. Besides, the deployment of the interpreter costs 3,365,098 gas, while deploying a single IFLOW node costs 721,049 gas.

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

Table 6: CATERPILLAR's interpretation-based engine: process instantiation and execution costs.

Process	Traces	Approach	Average Cost	
			Instant.	**Exec.**
Invoicing	5316	Default	1,089,000	383,109
		Opt- CF	807,123	297,351
		C- Caterp	**2,830,063**	**1,088,315**
		I- Caterp	**543,503**	**652,784**
Supply chain	62	Default	304,084	281,206
		Opt- CF	298,564	272,186
		C- Caterp	**1,100,590**	**566,861**
		I- Caterp	**434,891**	**418,259**
Incident mgmt.	124	Default	365,207	185,680
		Opt- CF	345,743	166,345
		C- Caterp	**1,119,803**	**324,420**
		I- Caterp	**496,038**	**273,811**
Insurance claim	279	Default	439,143	552,274
		Opt- CF	391,510	514,712
		C- Caterp	**1,338,152**	**1,235,617**
		I- Caterp	**500,614**	**992,461**

Table 6 shows the gas consumption observed in the experiments. For comparison, we used three baselines, in addition to the approach described by this paper (labelled as *I- Caterp* in Table 6). The first baseline (labelled *Default*) corresponds to the approach presented in [163] which compiles the control-flow perspective into a smart contract that also stores one boolean variable per (binary) decision gateway in order to determine which conditional flow should be selected. The second baseline (labelled *Opt- CF*) is similar to *Default* but it uses reduction rules to simplify the control-flow structure of the process model prior to compilation [56].[7] These baselines focus on the control-flow perspective. They do not handle the data and resource perspective (i.e. storing data attributes and managing work-items). The fourth baseline, named *C- Caterp* corresponds to the compilation-based version of CATERPILLAR described in the previous chapter, which provides a more advanced architecture, capable of handling the data and resource perspectives. In all the cases, Table 6 shows the average costs to instantiate the processes and to execute a trace in the event log.

[7] In the *Default* and *Opt- CF* approaches, one smart contract is deployed per process instance. In [56], a second optimized approach (*Opt-Full*) is proposed wherein all instances of a process are executed by a single smart contract, thus leading to lower instantiation costs. However, this approach cannot be extended to deal with data and resources because, when data is involved, one contract per process instance is needed to hold the instance data. Given this fundamental limitation, we exclude this approach from this comparison.

Table 6 shows that, in all the cases, the interpreter consumes significantly less gas than the compilation-based engine of CATERPILLAR. This result was expected given that: (1) the deployment costs are amortized as the number of process instances grows, given that the interpreter and the IFLOW structure are reused, and (2) redundancies in the code generation, present in compiled approaches, are eliminated, resulting in the reduction of the size of the smart contracts. Table 6 also shows the costs of the interpreter are relatively close to the approaches represented by *Default* and *Opt- CF*. Although in most of the cases the interpreter consumed more gas, it is worth noting that the comparison is not straightforward as *Default* and *Opt- CF* mainly focus on the control-flow perspective. In contrast, the interpreter implements a more advanced and flexible architecture which handles the three process perspectives, and also more advanced control-flow elements like sub-processes, multi-instances, and event propagation.

Like in the compilation-based engine, the process execution on the interpreter may face latency issues. However, these issues come from the blockchain platform (see Section 4.4.3).

5.5. Summary

This chapter addressed our second research question: *How can collaborative processes involving mutually untrusted parties be flexibly and cost-efficiently executed on a blockchain platform?* Accordingly, we presented an interpreted blockchain-based execution engine for collaborative business processes. Unlike previous approaches that rely on compilation of BPMN models into smart contracts, the proposed engine relies on a BPMN interpreter that takes as input a space-optimized representation of process models. This design reduces the costs of deployment since the smart contract encoding the interpreter only needs to be deployed once. It also allows participants to make changes to the process model in a way that these changes can be applied both to new process instances and to already running instances.

The interpreter has been integrated into the CATERPILLAR system, such that CATERPILLAR supports both a compiled and an interpreted execution approach. An empirical evaluation shows that the interpreted approach is more cost-efficient than the compiled one. Besides, despite supporting all three process modelling perspectives (control-flow, data, and resources), the costs of the CATERPILLAR interpreter are comparable to those of existing baselines that only support the control-flow perspective.

6. CONTROLLED FLEXIBILITY IN BLOCKCHAIN-BASED BUSINESS PROCESSES

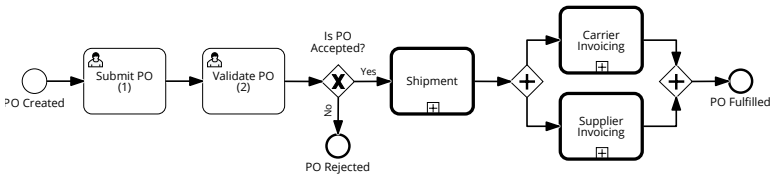
Chapter 5 presented a BPMN interpreter, which allows a flexible execution of processes such that they can be updated at runtime. However, flexibility may introduce trust issues if no adequate access control mechanism to restrict how a participant alters the process execution. Accordingly, in this chapter, we focus on the research question **RQ3**: *Which access control mechanisms would allow us to capture the wide range of dynamic binding and rebinding scenarios found for collaborative processes between mutually untrusted parties?* As a solution, we propose two mechanisms for controlled flexibility on collaborative processes: (i) to bind actors to roles dynamically, and (ii) for consensus-based control-flow flexibility.

The chapter is structured as follows. Section 6.1 describes the role-binding model and its associated policy language. Next, Section 6.2 presents the control-flow flexibility mechanisms and the associated agreement model and policy language. Section 6.3 discusses the semantics of the proposed policy languages and presents a verification approach to detect circular dependencies in role binding policies. Finally, Section 6.4 discusses the implementation and experimental evaluation, while Section 6.5 summarises the key aspects of our proposal.

6.1. Dynamic Role Binding

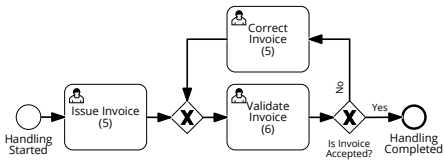
The starting point of the proposed approach is a (collaborative) business process model where each task is associated with a role. For a given process instance (herein called a *case*), each role may be assigned to at most one actor. An actor has an identity (e.g. a blockchain account) and may represent a user, a group, an organization, a system or a device. As a running example, Figure 23 shows a BPMN model of another 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 and the carrier's invoices respectively.

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 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



(1) Root process: Order-to-Cash

(2) Sub-process: Shipment



(3) Sub-process: Invoicing

Figure 23: Running example: (1) An *Order-to-cash* process linked, via call activities, to two reusable sub-processes; (2) *Shipment* and (3) *Invoicing*.

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.

6.1.1. 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 [136]. 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. Figure 24 shows an extract of the grammar of the policy language in Backus Naur form (BNF).[1]

Listing 12 shows a policy for the model in Figure 23. The policy states that

[1]For conciseness, some general details (e.g., path expressions to refer to nested sub-processes) are omitted and can be found at `http://git.io/caterpillar`.

⟨*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 24: BNF grammar describing the basic statement syntax of a binding policy.

```
{
 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 12: Binding Policy to control the execution of the processes modelled in Figure 23.

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 the endorsement of the Supplier and Customer, and reciprocally for the Invoicee. Meanwhile, under the Supplier 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 also be written 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 voter is specified, all participants that are currently assigned to a role are voters. Ratio expressions are less restrictive than boolean expressions as they rely on the amount instead of who is casting the votes. An example using ratio expressions can be found in Section 6.2.

6.1.2. 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 Figure 25. 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

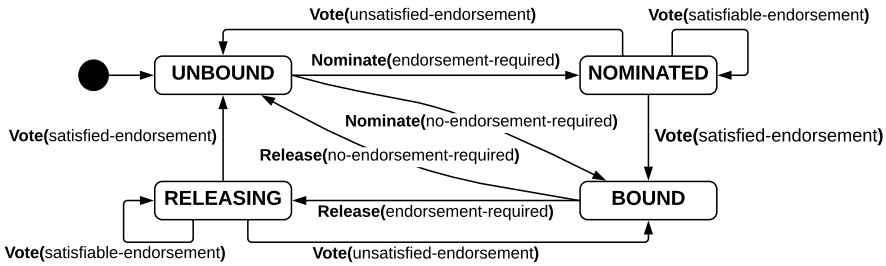


Figure 25: Life-cycle of a role within a case.

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 Figure 9 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.

The binding of 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` 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[2]:

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

[2] if an only if

p-case,

(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.[3] Besides, if a biding 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 as 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 biding 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 a nomination/release of nominee in p-case iff:

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

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

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

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, find the closest case scope, namely S, 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.

[3] 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

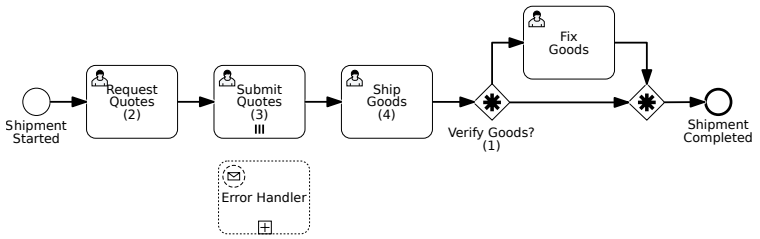


Figure 26: A more flexible variant of the sub-process GOODS SHIPMENT displayed in Figure 23.

6.2. 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 specific 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 Figure 23, one of the candidates made a mistake when submitting the quotes. Accordingly, the supplier would like to allow him to fix it before making the final decision, but no task exists to that end in the control-flow, and allowing to roll back the process state could introduce inconsistencies. However, a late-binding of a non-interrupting event sub-process (see Figure 26), running in parallel with the current process case [58], 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 that it will introduce some extra and unnecessary responsibilities to the other participants. Instead, an agreement policy can include a statement granting that in the sub-process SHIPMENT, a candidate can link a sub-process if the supplier agrees.

Overall, our initial proposal for the agreements policies supports flexibility by underspecification. Specifically, the process definition may be incomplete. However, at runtime, the participants provide the concrete realization of a process instance relaying on the late modelling and late-binding of sub-processes, e.g., on call-activities and event-sub-processes [140]. Accordingly, regarding the taxonomy of flexibility presented by Reichert and Weber [128], the agreement policies advocates for looseness and adaptation. Thus, in addition to allowing incomplete processes at design time, they may be adapted to handle exceptions at runtime, and the participants can decide alternative paths dynamically.

6.2.1. Agreement Policies on Control-Flow

An agreement policy 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 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/executes 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 a sub-process call activity. Figure 27 illustrates an extract of the BNF grammar of the policy language.[4]

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

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

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

Figure 27: 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.

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 [1, 2], pockets of flexibility [137, 139], late binding and late modelling [140, 154, 160]. 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

[4] Some details (e.g., path expressions to refer to nested sub-processes) are omitted for conciseness and can be found at `http://git.io/caterpillar`.

```
{
  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 13: Agreement Policy to support the execution of the processes modeled in Figures 23 and 26.

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. 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 13 illustrates an agreement policy extending the role-binding policy in Listing 12, and related to the models in Figures 23 and 26. 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, the supplier could decide, for example, between the sub-process in Figure 23 (2), or the one in Figure 26 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 sub-process 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 SHIP-

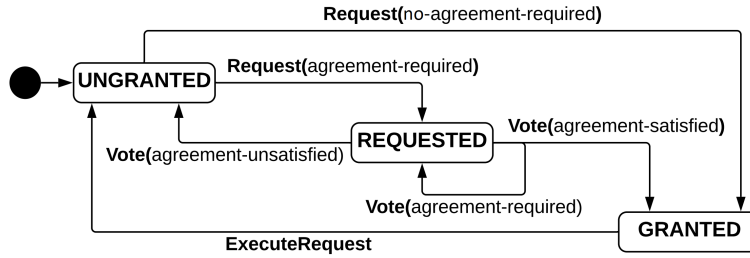


Figure 28: Life-cycle of an action to be performed at runtime.

MENT sub-process in Figure 26, 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 either at the supplier or 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.

6.2.2. 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 (cf. Chapters 4 and 5), 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 instantiate the sub-process and the hash identifying the compilation artefacts of the sub-process 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 life-cycle* depicted in Figure 28. 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 life-cycle. 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 life-cycle.

6.3. 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 [105], herein called a *nomination net*. A Petri net (P,T,F) consists of a set P of places (circles), a set T of transitions (squares) disjoint from P and a set of directed arcs $F \subseteq (P \times T) \cup (T \times P)$. Transitions represent activities in the model, while places allow us to model the state of a process. As such, places may contain tokens (black dots) that determine the state of the process. Transitions are enabled if at least one token exists in each of the incoming places, i.e., connected (via an outgoing arc) to the transition. Enabled transitions can be fired (executed), thus consuming the tokens in their incoming places, and producing new tokens in the outgoing places, i.e., those connected to the transition via incoming arcs. Accordingly, to guarantee that a role-binding policy is consistent, we only need to verify that all the transactions in the *nomination net* can be fired.

Given the nomination net of a binding policy, we map the problem of checking policy consistency to a problem of reachability analysis over Petri nets [158]. Algorithm 3 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 Figure 29. 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 Figure 29 is given in Figure 30. Given this input, the algorithm will produce as output the nomination net in Figure 31.

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 colour blue in Fig-

Algorithm 3 Construction of the Nomination Net for a given Binding Policy

1: **function** CONSTRUCTNOMINATIONNET(R, BP)
2: RNets $\leftarrow \emptyset$
▷ Step 1: Build a Petri net for each role
3: **for each** role $r \in$ R **do**
4: RNets $\leftarrow$ 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** NNet $= \langle P, T, F, M_0 \rangle$ **in**
6: $P \leftarrow \bigcup_{r \in \text{R}} \mathscr{P}(\text{RNets}[r])$
7: $T \leftarrow \bigcup_{r \in \text{R}} \mathscr{T}(\text{RNets}[r])$
8: $F \leftarrow \bigcup_{r \in \text{R}} \mathscr{F}(\text{RNets}[r])$
9: $M_0 \leftarrow \emptyset$
10:
▷ Step 3: Wire up operation NOMINATE
11: **for each** $\langle r_{nr}, r_{ne}, \_ \rangle \in$ BP **do**
12: **select** $b_{nr} \in \mathscr{P}(\text{RNets}[nr])$
13: **select** $nm_{r_{ne}} \in \mathscr{T}(\text{RNets}[ne])$
14: $\mathscr{F}(\text{NNet}) \leftarrow \mathscr{F}(\text{NNet}) \cup \{(b_{r_{nr}}, nm_{r_{ne}}), (nm_{r_{ne}}, b_{r_{nr}})\}$
▷ Step 4: Wire up operation ENDORSE
15: **for each** $\langle r_{nr}, r_{ne}, eex \rangle \in$ BP **such that** $eex \neq \perp$ **do**
16: $\mathscr{P}(\text{NNet}) \leftarrow \mathscr{P}(\text{NNet}) \cup \{disj_{r_{ne}}, eex_{r_{ne}}\}$
17: $\mathscr{F}(\text{NNet}) \leftarrow \mathscr{F}(\text{NNet}) \cup \{(nm_{r_{ne}}, disj_{r_{ne}}), (eex_{r_{ne}}, en_{r_{ne}})\}$
18: **for each** conj $\in eex$ **do**
19: $\mathscr{T}(\text{NNet}) \leftarrow \mathscr{T}(\text{NNet}) \cup \{eex_{conj}\}$
20: $\mathscr{F}(\text{NNet}) \leftarrow \mathscr{F}(\text{NNet}) \bigcup_{r \in conj \wedge b_r \in \mathscr{P}(\text{RNets}[r])} \left\{ \begin{array}{l} (b_r, eex_{conj}), (eex_{conj}, b_r), \\ (disj_{r_{ne}}, eex_{conj}) \end{array} \right\}$
▷ Step 5: Update NNet's initial marking
21: **let** $r_{cc} \in R$: r_{cc} be case creator **in**
22: Ps $\leftarrow \{u_r \mid r \in R \setminus \{r_{cc}\} \wedge u_r \in \mathscr{P}(\text{NNet}[r])\} \cup \{b_{r_{cc}} \mid b_{r_{cc}} \in \mathscr{P}(\text{NNet}[r_{cc}])\}$
23: $M_0(\text{NNet})(p) = \left\{ \begin{array}{ll} 1 & \text{if } p \in Ps \\ 0 & \text{Otherwise} \end{array} \right.$
24:
25: **return** NNet

ure 31. 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 life-cycle 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 similarly. 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 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

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

Figure 29: Sample binding policy

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

Figure 30: Symbolic representation of the binding policy in Figure 29

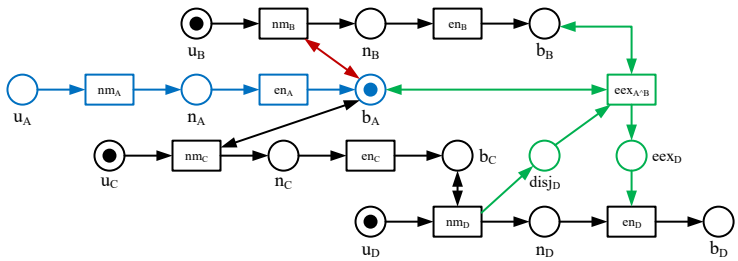


Figure 31: Nomination net for binding policy in Figure 29

and the transition nm_B in Figure 31, 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 considered as a single conjunction set. We consider two additional cases: (1) no endorsement condition is specified (represented by $\bot$), 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 Figure 32 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. 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 Figure 32). 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 Figure 32 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

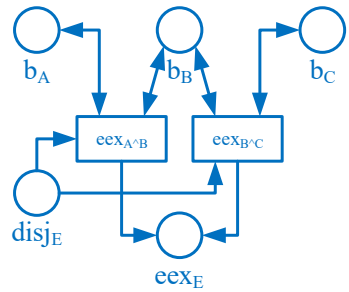


Figure 32: Net encoding condition $(A \wedge B) \vee (B \wedge C)$

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

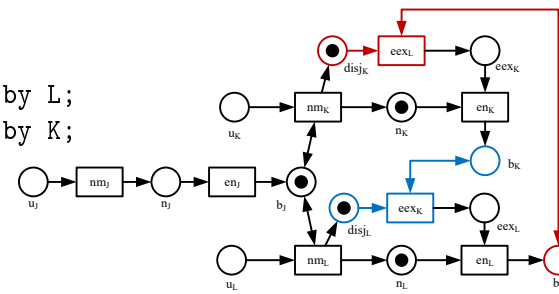


Figure 33: Binding policy with circular dependency and its nomination net

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 29-31, 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 Figure 31.

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 the `case-creator` are bound. In other words, there is no deadlock preventing a role from being bound. Figure 33 shows a binding policy with a circular dependency, leading to a deadlock in the corresponding nomination net. Figure 33 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 all the required actors 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.

6.4. 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 (cf. Chapters 4 and 5). 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.

6.4.1. Compiling Role-Binding Policies into Smart Contracts

From a process model and a policy specification, the policy compiler generates a smart contract (named BINDINGPOLICY) to encode the role-binding policy and a smart contract (TASKROLEMAP) to encode the task-role relations in the process model. The BINDINGPOLICY contract encodes the logic of who can nominate and release each role and the binding and endorsement constraints for each role. A third contract (BINDINGACCESSCONTROL) implements the runtime operations sketched in Section 6.1. 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 Figure 25. 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.

The class diagram in Figure 34 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.[5] Besides, the contract BINDINGACCESSCONTROL provides the functions `findState`, `findRole` and `linkTaskToRole` to query the

[5]Note that implementing the bit-sets 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.

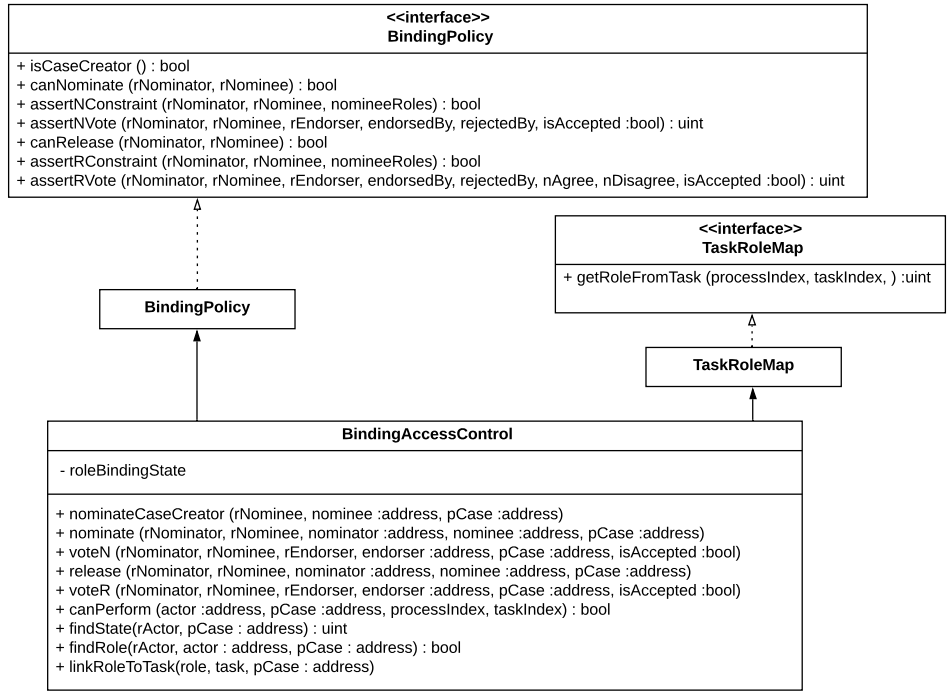


Figure 34: Class diagram of the smart contracts derived from the role-binding policies.

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 similarly.

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:

```
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 at least one role rejected the nomination in each conjunction set, i.e., for each `CS`, `(rejectedBy | endorserRole) & CS != 0`,
3. NOMINATED if none of the conditions 1. and 2. are satisfied yet, i.e., at least one conjunction set with no rejections and with roles pending to vote exists.

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.

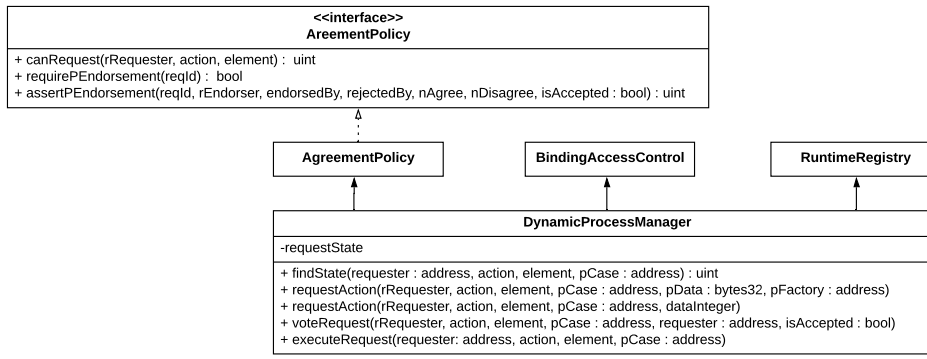


Figure 35: Class diagram of the smart contracts derived from the agreement policies.

6.4.2. Compiling Agreement Policies into Smart Contracts

Figure 35 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 Section 6.2. 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 bit-sets.

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 who 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 the 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 (cf. Chapters 4 and 5), to update the control-flow. 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`. The generation of the function `assertPEndorsement` is done in a similar way 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;
}
```

6.4.3. Experimental Setup

We conducted an empirical 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?[6] 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 6.4.1 that the costs derived from a role-binding policy depend on the number of roles to nominate and the number of conjunction 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

[6] In Ethereum, gas is linearly related to throughput, see Section 2.3. So by answering this question we also indirectly answer the related throughput question.

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 6.4.1 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, 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 7 resumes 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 run-

Table 7: 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.

time operations via CATERPILLAR's REST API. For each transaction included in the blockchain, CATERPILLAR sends metadata that includes block number, consumed gas, transaction hash which is collected and assessed by the replayer. For the experimentation, we run the Node.js based Ethereum client named ganache-cli[7] which simulates a full client for developing and testing purposes on Ethereum.

6.4.4. 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 Figure 36. It shows that deployment costs increase quasi-linearly with the size and complexity of the policy[8]. 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, Figure 37 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 complex-

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

[8] Note that figures 36 and 37 displays not the entire cost of the policies, but the growth costs derived from the instructions assessed in each experiment.

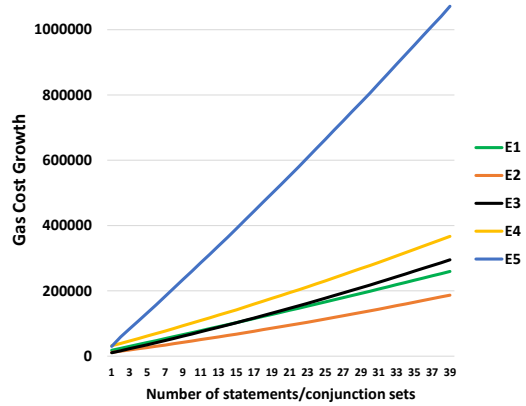


Figure 36: Growth of deployment costs with size of a role-binding policy.

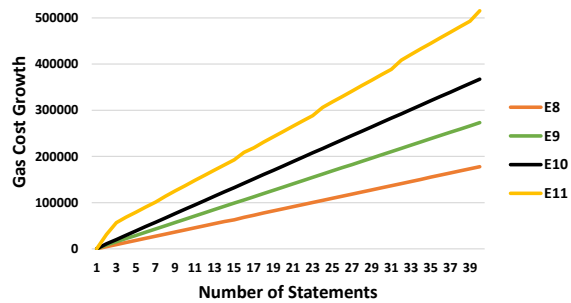


Figure 37: Growth of deployment costs with size of an agreement policy.

ity 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 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 be-

Table 8: 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 9: 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

cause 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%.

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 behaviour exists for binding and endorsement constraints. From the perspective of the algorithmic computational complexity [39], 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 8 shows the min, max, and average costs to perform the nominate and vote operations in experiments E1-E5. Similarly, Table 9 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 more costly than nominating in a role-binding policy. The later 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 the 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.

There is essential to consider that the smart contracts derived from the role-binding and agreement policies can be reused (deployed only once). In contrast, the contracts handling the process models typically requires a new deployment for each process case. Accordingly, several executions of a process model lead to amortize the deployment costs of the policies (if reused).

Estimating the overhead added by the policies to the process execution is not straightforward due to the many combinations and scenarios coming 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 [88, 90]. 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 55260 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 [90] and interpreted [88] approaches, respectively. We also calculated minimum and maximum deployment and execution bound for the role-binding policies putting together 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 10 illustrates the values in which deployment and execution are expected to range. Besides, the letters C and I, following the process models label, corresponding to the variants compiled and interpreted, respectively. The total

Table 10: Comparison of deployment and execution costs between role-binding policies and business process models.

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

costs[9] without reusing the policies would add an overhead between 1,735,379 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 to provide a full picture of the worst scenario. Instead, for example, the traces in the event log of the invoicing process includes 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 10). 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 dependant to a specific process case, thus performed only once and reused.

Figure 38 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 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.

[9]The total costs include the fixed cost contracts BINDINGACCESSCONTROL and TASKROLEMAP, and the corresponding BINDINGPOLICY (min or max accordingly). Besides, the execution costs of the smart contract TASKROLEMAP are included in BINDINGACCESSCONTROL, i.e., from the execution of the function `canPerform`.

Figure 38: Variation of the amortized deployment and execution costs of role-binding policies by reusing them across different process cases.

6.5. Summary

In this chapter, we addressed our third, and last, research question: *Which access control mechanisms would allow us to capture the wide range of dynamic binding and rebinding scenarios found for collaborative processes between mutually untrusted parties?* As a solution, we presented an approach to extend blockchain-based collaborative process execution platforms with controlled flexibility mechanisms. Specifically, our contributions are:

1. A role-binding model and an associated binding policy language to allow 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 (and therefore, throughput) 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.

7. CONCLUSION AND FUTURE WORK

7.1. Summary of contributions

This thesis demonstrates how to combine the high-level abstractions of business process management systems with the trust-enhancing capabilities of blockchain technology in order to support the execution of collaborative business processes involving untrusted parties. To that end, we developed an open-source blockchain-based BPMN execution engine named CATERPILLAR.

To the best of our knowledge, CATERPILLAR is the first blockchain-based process execution engine capable of handling process models with sub-processes, as well as advanced BPMN constructs such as boundary events and multi-instance activities. Also, CATERPILLAR is the first system that executes the entire collaborative process on the blockchain in the sense that all the state of the process instances and their links are maintained on-chain, thus encoding all the control-flow, data-flow and resource logic in smart contracts. Similarly, CATERPILLAR is the first system that does not assume that the parties in the process use message exchanges for coordination. However, instead, the parties use the blockchain as the only coordination mechanism.

Specifically, this thesis makes three contributions to the field of automated execution of collaborative business processes on the blockchain. The ***first contribution*** of the thesis is an engine which compiles hierarchical business processes enhanced with data and resource constraints into smart contracts. The engine supports deployment of the smart contracts in the (Ethereum) blockchain and monitoring the process execution through a set of on-chain and off-chain components. Accordingly, we discussed the rationale behind each component of the architecture of CATERPILLAR's compilation-based engine, and how they interact as a whole BPMS which require no trust among the participants. In addition, we described how to translate a wide set of BPMN elements into smart contracts in Solidity as implemented by the CATERPILLAR's compiler. Finally, we measured the costs to deploy the smart contracts and to execute the process tasks. As a result, the empirical evaluation showed the feasibility of the compilation-based engine to handle realistic process models.

Compiled approaches to blockchain-based process execution use the immutability of the smart contracts to enforce trust in the process execution, i.e., participants cannot change the process execution on their own behalf. However, this immutability constitutes a limitation for processes that may be subject to changes at runtime. Indeed, when following a compiled approach, even a minimal change to a process model requires the redeployment of all the smart contracts, which leads to efficiency issues.

In this context, our ***second contribution*** proposes an interpreted execution to mitigate the inflexibility and efficiency issues inherent in compiled approaches. Our interpreter supports the same BPMN constructions as the CATERPILLAR's

compilation-based engine. However, the proposed interpreter relies on space-optimized representations of process models using bitmap-based encodings that are stored and handled by dynamic data structures. The interpreter is embedded in a modular multi-layered architecture integrated into the CATERPILLAR system. This design reduces the costs of deployment since the smart contract encoding the interpreter only needs to be deployed once. It also allows participants to make changes to the process model at runtime. An experimental evaluation shows that the CATERPILLAR interpretation-engine achieves comparable or lower costs relative to existing compiled solutions. We compared the throughput of the CATERPILLAR interpreter with several baselines which focus only on the control flow-perspective. Thus, although the comparison is not straightforward, as the functionality provided by CATERPILLAR is more sophisticated, i.e., it includes data-flow and resources, it exposes the efficiency improvements introduced by the interpreter.

The interpreted approach to collaborative business processes execution conveniently allows us to deploy and change processes dynamically. However, it may lead to trust issues without an adequate control mechanism to update the process at runtime. To address this limitation, our ***third contribution*** proposes two mechanisms for controlled flexibility on collaborative processes. First, we designed a model for dynamic binding of actors to roles in collaborative processes and an associated binding policy specification language. This binding-model offers a mechanism to dynamically control the access of process participants, which is suitable for both compiled and interpreted approaches. The proposed model is endowed with a Petri net semantics that enables consistency verification of the policies. The second is a model for consensus-based control-flow flexibility, which is more suitable for interpreted approaches as it focuses on controlling the updating of the process at runtime. Expressly, participants in a process can collectively agree on how to steer the business process within the boundaries defined by control-flow agreement policies. For both models, we proposed approaches that translate policy specifications into smart contracts for enforcement. Finally, the experimental evaluation shows that the cost of policy enforcement increases linearly with the number of roles, control-flow elements, and policy constraints.

All the contributions discussed in this thesis are implemented in the CATERPILLAR system. The source code of CATERPILLAR is available in a public repository under the SD 3-clause “New" or “Revised" (see Appendix A).

7.2. Future work

The contributions described in this thesis open up multiple possibilities for future work outlined in the following paragraphs.

In this thesis, we represented process data code snippets written in the Solidity language, which are embedded in the process models. Further research needs to focus on high-level representations of the data and conditions for blockchain-

based collaborative processes. Among the challenges regarding data access and representation, for example, encrypted data provides confidentiality [29], but it reduces enforceability of the process as no operation can be performed with such data [79]. This trade-off between confidentiality and enforceability becomes even more critical in the presence of dynamic scenarios. In such cases, the process participants must reach consensus at runtime about how and by whom the process data can be accessed/updated. Other challenges come when deciding how to selectively share data among participants, as well as when performing trusted operations on secret data, under the performance and scalability limitations of the blockchains. In addition, the immutability of the transactions in the blockchain may introduce issues when revoking privileges. For example, process participants may have access to encrypted data forever once they own the decryption key, even after leaving the organization. Designing models for shared data to overcome the latest challenges is a venue for future work.

The flexibility mechanisms proposed in this thesis 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, to perform specific tasks of the process, one expects this participant to have access to the data required to fulfil 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 with this role. One 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 fact that our approach for flexibility allows participants to dynamically update a process model raises the question of how to ensure that the already-running instances do not end up in an inconsistent state after a process model change. For example, replacing a pair of XOR gateways with AND gateways may put some instances in an inconsistent state, possibly leading to a deadlock. A direction for future work is to adapt existing approaches for consistency verification of dynamic process model changes to this setting [128]. In this setting, the verification and monitoring of blockchain-based processes and the applicability of process mining techniques mostly remains as an unexplored research area. Although some nascent works already extract event logs for process mining from the blockchain [77, 103], they are mainly platform dependent; thus, further research is still required.

Finally, while the experimental evaluation shows that the CATERPILLAR compiler can handle realistic process models, it also suggests that the approach would not scale to extensive process models with hundreds or thousands of elements when running on a public blockchain. Higher scalability could be achieved by using consortium blockchain technology, such as Hyperledger or Ethereum

Proof-of-Authority consensus, as well as exploring other blockchains architectures which support much higher throughput [162]. Investigating the performance of the CATERPILLAR approach on different blockchain platforms and configurations is a direction for future work. Another would focus on the portability of the CATERPILLAR system, enhancing it to allow collaborative processes to be executed across multiple blockchains platforms [50].

BIBLIOGRAPHY

[1] Michael Adams, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, and David 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*, pages 95–112, 2007.

[2] Michael Adams, Arthur H.M. ter Hofstede, David Edmond, and Wil 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*, pages 291–308, 2006.

[3] Kevin Andrews, Sebastian Steinau, and Manfred Reichert. Enabling runtime flexibility in data-centric and data-driven process execution engines. *Information Systems*, page 101447, 2019.

[4] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web services, 2003. `http://cliplab.org/Projects/S-CUBE/papers/andrews03:BPELWS-1.1.pdf` – last accessed 2020-03-15.

[5] Ali Arsanjani, Liang-Jie Zhang, Michael Ellis, Abdul Allam, and Kishore Channabasavaiah. S3: A service-oriented reference architecture. *IT Professional*, 9(3):10–17, 2007.

[6] Larissa Auberger and Matthias Kloppmann. Digital process automation with BPM and blockchain, part 1: Combine business process management and blockchain, 2017. `https://www.ibm.com/developerworks/library/mw-1705-auberger-bluemix/1705-auberger.html` – last accessed 2020-03-15.

[7] Adriano Augusto, Raffaele Conforti, Marlon Dumas, Marcello La Rosa, and Giorgio Bruno. Automated discovery of structured process models: Discover structured vs. discover and structure. In *Conceptual Modeling - 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings*, pages 313–329, 2016.

[8] Clara Ayora, Victoria Torres, Jose Luis de la Vara, and Vicente Pelechano. Variability management in process families through change patterns. *Inf. Softw. Technol.*, 74:86–104, 2016.

[9] Clara Ayora, Victoria Torres, Barbara Weber, Manfred Reichert, and Vicente Pelechano. VIVACE: A framework for the systematic evaluation of

variability support in process-aware information systems. *Inf. Softw. Technol.*, 57:248–276, 2015.

[10] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. Medrec: Using blockchain for medical data access and permission management. In *2nd International Conference on Open and Big Data, OBD 2016, Vienna, Austria, August 22-24, 2016*, pages 25–30, 2016.

[11] Alistair P. Barros, Marlon Dumas, and Arthur H.M. ter Hofstede. Service interaction patterns. In *Business Process Management, 3rd International Conference, BPM 2005, Nancy, France, September 5-8, 2005, Proceedings*, pages 302–318, 2005.

[12] Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*, pages 494–509, 2017.

[13] Anne Baumgrass, Claudio Di Ciccio, Remco M. Dijkman, Marcin Hewelt, Jan Mendling, Andreas Meyer, Shaya Pourmirza, Mathias Weske, and Tsun Yin Wong. GET controller and UNICORN: event-driven process execution and monitoring in logistics. In *Proceedings of the BPM Demo Session 2015 Co-located with the 13th International Conference on Business Process Management (BPM 2015), Innsbruck, Austria, September 2, 2015*, pages 75–79, 2015.

[14] Daniel Beimborn and Nils Joachim. The joint impact of service-oriented architectures and business process management on business process quality: an empirical evaluation and comparison. *Inf. Syst. E-Business Management*, 9(3):333–362, 2011.

[15] Boualem Benatallah, Quan Z. Sheng, Anne H. H. Ngu, and Marlon Dumas. Declarative composition and peer-to-peer provisioning of dynamic web services. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 297–308, 2002.

[16] Elisa Bertino, Elena Ferrari, and Vijayalakshmi Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Trans. Inf. Syst. Secur.*, 2(1):65–104, 1999.

[17] Saïda Boukhedouma, Zaia Alimazighi, and Mourad Chabane Oussalah. Evolution of inter-organizational workflows: The case-transfer pattern. In *4th IEEE International Colloquium on Information Science and Technology, CiSt 2016, Tangier, Morocco, October 24-26, 2016*, pages 235–242, 2016.

[18] Ruth Breu, Schahram Dustdar, Johann Eder, Christian Huemer, Gerti Kappel, Julius Köpke, Philip Langer, Jürgen Mangler, Jan Mendling, Gustaf Neumann, Stefanie Rinderle-Ma, Stefan Schulte, Stefan Sobernig, and Bar-

bara Weber. Towards living inter-organizational processes. In *IEEE 15th Conference on Business Informatics, CBI 2013, Vienna, Austria, July 15-18, 2013*, pages 363–366, 2013.

[19] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: an introduction. *R3 CEV, August*, 1:15, 2016.

[20] Antonio Bucchiarone, Marco Pistore, Heorhi Raik, and Raman Kazhamiakin. Adaptation of service-based business processes by context-aware replanning. In *2011 IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2011, Irvine, CA, USA, December 12-14, 2011*, pages 1–8, 2011.

[21] Laurent Bussard, Anna Nano, and Ulrich Pinsdorf. Delegation of access rights in multi-domain service compositions. *Identity in the Information Society*, 2(2):137–154, 2009.

[22] Vitalik Buterin. Ethereum and Oracles, 2014. `https://blog.ethereum.org/2014/07/22/ethereum-and-oracles/` – last accessed 2020-03-15.

[23] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *white paper*, 3(37), 2014.

[24] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[25] Cristina Cabanillas, Manuel Resinas, Adela del-Río-Ortega, and Antonio Ruiz Cortés. Specification and automated design-time analysis of the business process human resource perspective. *Inf. Syst.*, 52:55–82, 2015.

[26] Cristina Cabanillas, Manuel Resinas, Jan Mendling, and Antonio Ruiz Cortés. Automated team selection and compliance checking in business processes. In *Proceedings of the 2015 International Conference on Software and System Process, ICSSP 2015, Tallinn, Estonia, August 24 - 26, 2015*, pages 42–51, 2015.

[27] Christian Cachin. Architecture of the hyperledger blockchain fabric. In *Workshop on distributed cryptocurrencies and consensus ledgers*, volume 310, page 4, 2016.

[28] Seraphin B. Calo, Dinesh C. Verma, Supriyo Chakraborty, Elisa Bertino, Emil Lupu, and Gregory H. Cirincione. Self-generation of access control policies. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies, SACMAT 2018, Indianapolis, IN, USA, June 13-15, 2018*, pages 39–47, 2018.

[29] Barbara Carminati, Christian Rondanini, and Elena Ferrari. Confidential business process execution on blockchain. In *2018 IEEE International Conference on Web Services, ICWS 2018, San Francisco, CA, USA, July 2-7, 2018*, pages 58–65, 2018.

[30] Fabio Casati, Stefano Ceri, Barbara Pernici, and Giuseppe Pozzi. Workflow evolution. *Data Knowl. Eng.*, 24(3):211–238, 1998.

[31] James F. Chang. *Business process management systems: strategy and implementation*. Auerbach Publications, 2016.

[32] Victor Chang, Yen-Hung Kuo, and Muthu Ramachandran. Cloud computing adoption framework: A security framework for business clouds. *Future Generation Comp. Syst.*, 57:24–41, 2016.

[33] Victor Chang, Robert John Walters, and Gary Wills. The development that leads to the cloud computing business framework. *Int J. Information Management*, 33(3):524–538, 2013.

[34] Claudio Di Ciccio, Alessio Cecconi, Marlon Dumas, Luciano García-Bañuelos, Orlenys López-Pintado, Qinghua Lu, Jan Mendling, Alexander Ponomarev, An Binh Tran, and Ingo Weber. Blockchain support for collaborative business processes. *Informatik Spektrum*, 42(3):182–190, 2019.

[35] Claudio Di Ciccio, Alessio Cecconi, Jan Mendling, Dominik Felix, Dominik Haas, Daniel Lilek, Florian Riel, Andreas Rumpl, and Philipp Uhlig. Blockchain-based traceability of inter-organisational business processes. In *Business Modeling and Software Design - 8th International Symposium, BMSD 2018, Vienna, Austria, July 2-4, 2018, Proceedings*, pages 56–68, 2018.

[36] Mamadou Lakhassane Cisse, Hanh Nhi Tran, Samba Diaw, Bernard Coulette, and Alassane Bah. Using patterns to parameterize the execution of collaborative tasks. In *28th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2019, Naples, Italy, June 12-14, 2019*, pages 106–111, 2019.

[37] Riccardo Cognini, Flavio Corradini, Stefania Gnesi, Andrea Polini, and Barbara Re. Business process flexibility - a systematic literature review with a software systems perspective. *Information Systems Frontiers*, 20(2):343–371, 2018.

[38] David Cohn and Richard Hull. Business artifacts: A data-centric approach to modeling business operations and processes. *IEEE Data Eng. Bull.*, 32(3):3–9, 2009.

[39] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[40] Michael Crosby, Pradan Pattanayak, Sanjeev Verma, and Vignesh Kalyanaraman. Blockchain technology: Beyond bitcoin. *Applied Innovation Review*, 2, 2016.

[41] Gaby Dagher, Jordan Mohler, Matea Milojkovic, and Praneeth Babu Marella. Ancile: Privacy-preserving framework for access control and interoperability of electronic health records using blockchain technology. *Sustainable cities and society*, 39:283–297, 2018.

[42] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias 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, pages 296–303, 2007.

[43] Sheng Ding, Jin Cao, Chen Li, Kai Fan, and Hui Li. A novel attribute-based access control scheme using blockchain for IoT. *IEEE Access*, 7:38431–38441, 2019.

[44] Markus Döhring, Hajo A. Reijers, and Sergey Smirnov. Configuration vs. adaptation for business process variant maintenance: An empirical study. *Inf. Syst.*, 39:108–133, 2014.

[45] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management, Second Edition*. Springer, 2018.

[46] Jacob Eberhardt and Stefan Tai. On or off the blockchain? insights on off-chaining computation and data. In *Service-Oriented and Cloud Computing - 6th IFIP WG 2.14 European Conference, ESOCC 2017, Oslo, Norway, September 27-29, 2017, Proceedings*, pages 3–15, 2017.

[47] Fatma Ellouze, Mohamed Amine Chaâbane, Rafik Bouaziz, and Eric Andonoff. Addressing inter-organisational process flexibility using versions: The VP2M approach. In *Tenth IEEE International Conference on Research Challenges in Information Science, RCIS 2016, Grenoble, France, June 1-3, 2016*, pages 1–12, 2016.

[48] Hamza Es-Samaali, Aissam Outchakoucht, and Jean Philippe Leroy. A blockchain-based access control for big data. *International Journal of Computer Networks and Communications Security*, 5(7):137–147, 2017.

[49] Ghareeb Falazi, Michael Hahn, Uwe Breitenbücher, and Frank Leymann. Modeling and execution of blockchain-aware business processes. *SICS Softw.-Intensive Cyber Phys. Syst.*, 34(2-3):105–116, 2019.

[50] Ghareeb Falazi, Michael Hahn, Uwe Breitenbücher, Frank Leymann, and Vladimir Yussupov. Process-based composition of permissioned and permissionless blockchain smart contracts. In *23rd IEEE International Enterprise Distributed Object Computing Conference, EDOC 2019, Paris, France, October 28-31, 2019*, pages 77–87, 2019.

[51] Walid Fdhila, Conrad Indiono, Stefanie Rinderle-Ma, and Manfred Reichert. Dealing with change in process choreographies: Design and implementation of propagation algorithms. *Inf. Syst.*, 49:1–24, 2015.

[52] Marco Franceschetti and Johann Eder. Dynamic service binding for time-aware service compositions. In *23rd IEEE International Enterprise Distributed Object Computing Workshop, EDOC Workshops 2019, Paris, France, October 28-31, 2019*, pages 146–151, 2019.

[53] Christopher Frantz and Mariusz 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*, pages 210–215, 2016.

[54] Erich Gamma. Helm. r., johnson, r., vlissides, j.: Design patterns: elements of reusable object-oriented software. *Addison Wesley Longman, Inc, January*, 1(5):1, 1995.

[55] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, pages 291–323, 2017.

[56] Luciano García-Bañuelos, Alexander Ponomarev, Marlon Dumas, and Ingo Weber. Optimized execution of business processes on blockchain. In *Business Process Management - 15th International Conference, BPM 2017, Barcelona, Spain, September 10-15, 2017, Proceedings*, pages 130–146, 2017.

[57] Paul Grefen and Stefanie Rinderle-Ma. Dynamism in inter-organizational service orchestration -an analysis of the state of the art. Technical report, Eindhoven University of Technology, 2016.

[58] Object Management Group. Business Process Model and Notation (BPMN), Version 2.0.2, 2014. `http://www.omg.org/spec/BPMN/2.0.2/` – last accessed 2020-03-15.

[59] Stephan Haarmann, Kimon Batoulis, Adriatik Nikaj, and Mathias Weske. DMN decision execution on the ethereum blockchain. In *Advanced Information Systems Engineering - 30th International Conference, CAiSE 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings*, pages 327–341, 2018.

[60] Stephan Haarmann, Kimon Batoulis, Adriatik Nikaj, and Mathias Weske. Executing collaborative decisions confidentially on blockchains. In *Business Process Management: Blockchain and Central and Eastern Europe Forum - BPM 2019 Blockchain and CEE Forum, Vienna, Austria, September 1-6, 2019, Proceedings*, pages 119–135, 2019.

[61] Zheng Haibei and Yin Xu. The architecture design of a distributed workflow system. In *2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering Science*, pages 9–12, 2012.

[62] Alena Hallerbach, Thomas Bauer, and Manfred Reichert. Capturing variability in business process models: the provop approach. *J. Softw. Maintenance Res. Pract.*, 22(6-7):519–546, 2010.

[63] Bernd Heinrich, Mathias Klier, and Steffen Zimmermann. Automated planning of process models: Design of a novel approach to construct exclusive choices. *Decision Support Systems*, 78:1–14, 2015.

[64] Jonathan Heiss, Jacob Eberhardt, and Stefan Tai. From oracles to trust-

worthy data on-chaining systems. In *IEEE International Conference on Blockchain, Blockchain 2019, Atlanta, GA, USA, July 14-17, 2019*, pages 496–503, 2019.

[65] Thomas T. Hildebrandt. Flexible, adaptable, and compliant business systems with dynamic condition response graphs. In *Proceedings of the International Workshop on Formal Methods for Analysis of Business Systems, ForMABS@ASE 2016, Singapore, Singapore, September 4, 2016*, page 1, 2016.

[66] David Hollingsworth and UK Hampshire. Workflow management coalition: The workflow reference model. *Document Number TC00-1003*, 19:16, 1995.

[67] Zhengxing Huang, Xudong Lu, and Huilong Duan. Mining association rules to support resource allocation in business process management. *Expert Syst. Appl.*, 38(8):9483–9490, 2011.

[68] Zhengxing Huang, Wil M.P. van der Aalst, Xudong Lu, and Huilong Duan. Reinforcement learning based resource allocation in business process management. *Data Knowl. Eng.*, 70(1):127–145, 2011.

[69] Richard Hull. Blockchain: Distributed event-based processing in a data-centric world: Extended abstract. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017*, pages 2–4, 2017.

[70] Richard Hull, Vishal S. Batra, Yi-Min Chen, Alin Deutsch, Fenno F. Terry Heath III, and Victor Vianu. Towards a shared ledger business collaboration language based on data- aware processes. In *Service-Oriented Computing - 14th International Conference, ICSOC 2016, Banff, AB, Canada, October 10-13, 2016, Proceedings*, pages 18–36, 2016.

[71] Mayssa Jemel and Ahmed Serhrouchni. Decentralized access control mechanism with temporal dimension based on blockchain. In *14th IEEE International Conference on e-Business Engineering, ICEBE 2017, Shanghai, China, November 4-6, 2017*, pages 177–182, 2017.

[72] Dimka Karastoyanova, Alejandro Houspanossian, Mariano Cilia, Frank Leymann, and Alejandro P. Buchmann. Extending BPEL for run time adaptability. In *Ninth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2005), 19-23 September 2005, Enschede, The Netherlands*, pages 15–26, 2005.

[73] Dimka Karastoyanova, Frank Leymann, Jörg Nitzsche, Branimir Wetzstein, and Daniel Wutke. Parameterized BPEL processes: Concepts and implementation. In *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, 2006, Proceedings*, pages 471–476, 2006.

[74] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain pro-

tocol. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, pages 357–388, 2017.

[75] Henry M. Kim and Marek Laskowski. Toward an ontology-driven blockchain design for supply-chain provenance. *Int. Syst. in Accounting, Finance and Management*, 25(1):18–27, 2018.

[76] Justus Klingemann. Controlled flexibility in workflow management. In *Advanced Information Systems Engineering, 12th International Conference CAiSE 2000, Stockholm, Sweden, June 5-9, 2000, Proceedings*, pages 126–141, 2000.

[77] Christopher Klinkmüller, Alexander Ponomarev, An Binh Tran, Ingo Weber, and Wil M.P. van der Aalst. Mining blockchain processes: Extracting process mining data from blockchain applications. In *Business Process Management: Blockchain and Central and Eastern Europe Forum - BPM 2019 Blockchain and CEE Forum, Vienna, Austria, September 1-6, 2019, Proceedings*, pages 71–86, 2019.

[78] Matthias Kloppmann, Dieter Koenig, Frank Leymann, Gerhard Pfau, Alan Rickayzen, Claus von Riegen, Patrick Schmidt, and Ivana Trickovic. WS-BPEL extension for people - BPEL4People. *Joint white paper, IBM and SAP*, 183:184, 2005.

[79] Julius Köpke, Marco Franceschetti, and Johann Eder. Balancing privity and enforceability of bpm-based smart contracts on blockchains. In *Business Process Management: Blockchain and Central and Eastern Europe Forum - BPM 2019 Blockchain and CEE Forum, Vienna, Austria, September 1-6, 2019, Proceedings*, pages 87–102, 2019.

[80] Markus Kradolfer and Andreas Geppert. Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In *Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems, Edinburgh, Scotland, USA, September 2-4, 1999*, pages 104–114, 1999.

[81] Ajay Krishna, Pascal Poizat, and Gwen Salaün. Checking business process evolution. *Science of Computer Programming*, 170:1 – 26, 2019.

[82] Jan Ladleif, Mathias Weske, and Ingo Weber. Modeling and enforcing blockchain-based choreographies. In *Business Process Management - 17th International Conference, BPM 2019, Vienna, Austria, September 1-6, 2019, Proceedings*, pages 69–85, 2019.

[83] Jooseok Lee, Seunghoon Lee, Jinwoo Kim, and Injun Choi. Dynamic human resource selection for business process exceptions. *Knowledge and Process Management*, 26(1):23–31, 2019.

[84] Jiankun Lei, Rufan Bai, Lipeng Guo, and Liang Zhang. Towards a scalable framework for artifact-centric business process management systems. In *Web Information Systems Engineering - WISE 2016 - 17th International*

Conference, Shanghai, China, November 8-10, 2016, Proceedings, Part II, pages 309–323, 2016.

[85] Guoli Li, Vinod Muthusamy, and Hans-Arno Jacobsen. A distributed service-oriented architecture for business process execution. *TWEB*, 4(1):2:1–2:33, 2010.

[86] Orlenys López-Pintado. Business process execution on blockchain. In *Proceedings of the Doctoral Consortium at the 30th International Conference on Advanced Information Systems Engineering (CAiSE 2018), Tallinn, Estonia, June 11-15, 2018*, volume 2114 of *CEUR Workshop Proceedings*, pages 10–18, 2018.

[87] Orlenys López-Pintado, Marlon Dumas, Luciano García-Bañuelos, and Ingo 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*, pages 399–414, 2019.

[88] Orlenys López-Pintado, Marlon Dumas, Luciano García-Bañuelos, and Ingo Weber. Interpreted execution of business process models on blockchain. In *23rd IEEE International Enterprise Distributed Object Computing Conference, EDOC 2019, Paris, France, October 28-31, 2019*, pages 206–215, 2019.

[89] Orlenys López-Pintado, Luciano García-Bañuelos, Marlon Dumas, and Ingo Weber. Caterpillar: A blockchain-based business process management system. In *Proceedings of the Demo Track and Dissertation Award of the 15th International Conference on Business Process Management (BPM 2017), Barcelona, Spain, September 13, 2017*, 2017.

[90] Orlenys López-Pintado, Luciano García-Bañuelos, Marlon Dumas, Ingo Weber, and Alexander Ponomarev. Caterpillar: A business process execution engine on the ethereum blockchain. *Softw., Pract. Exper.*, 49(7):1162–1193, 2019.

[91] Qinghua Lu and Xiwei Xu. Adaptable blockchain-based systems: A case study for product traceability. *IEEE Software*, 34(6):21–27, 2017.

[92] Yahui Lu, Li Zhang, and Jiaguang Sun. Task-activity based access control for process collaboration environments. *Comput. Ind.*, 60(6):403–415, 2009.

[93] Mingxin Ma, Guozhen Shi, and Fenghua Li. Privacy-oriented blockchain-based distributed key management architecture for hierarchical access control in the iot scenario. *IEEE Access*, 7:34045–34059, 2019.

[94] Mads Frederik Madsen, Mikkel Gaub, Tróndur Høgnason, Malthe Ettrup Kirkbro, Tijs Slaats, and Søren Debois. Collaboration among adversaries: distributed workflow execution on a blockchain. In *Symposium on Foundations and Applications of Blockchain, Proceedings*, pages 8–15, 2018.

[95] Damiano Di Francesco Maesa, Paolo Mori, and Laura Ricci. Blockchain based access control. In *Distributed Applications and Interoperable Systems - 17th IFIP WG 6.1 International Conference, DAIS 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings*, pages 206–220, 2017.

[96] Damiano Di Francesco Maesa, Paolo Mori, and Laura Ricci. A blockchain based approach for the definition of auditable access control systems. *Computers & Security*, 84:93–119, 2019.

[97] Andrea Marrella, Alessandro Russo, and Massimo 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*, pages 268–286, 2012.

[98] Daniel Martin, Daniel Wutke, and Frank Leymann. A novel approach to decentralized workflow enactment. In *12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, 15-19 September 2008, Munich, Germany*, pages 127–136, 2008.

[99] Jan Mendling, Ingo Weber, Wil M.P. van der Aalst, Jan vom Brocke, Cristina Cabanillas, Florian Daniel, Søren Debois, Claudio Di Ciccio, Marlon Dumas, Schahram Dustdar, Avigdor Gal, Luciano García-Bañuelos, Guido Governatori, Richard Hull, Marcello La Rosa, Henrik Leopold, Frank Leymann, Jan Recker, Manfred Reichert, Hajo A. Reijers, Stefanie Rinderle-Ma, Andreas Solti, Michael Rosemann, Stefan Schulte, Munindar P. Singh, Tijs Slaats, Mark Staples, Barbara Weber, Matthias Weidlich, Mathias Weske, Xiwei Xu, and Liming Zhu. Blockchains for business process management - challenges and opportunities. *ACM Trans. Management Inf. Syst.*, 9(1):4:1–4:16, 2018.

[100] Lucie Mercenne, Kei-Leo Brousmiche, and Elyes Ben Hamida. Blockchain studio: A role-based business workflows management system. In *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 1215–1220, 2018.

[101] Giovanni Meroni, Pierluigi Plebani, and Francesco Vona. Trusted artifact-driven process monitoring of multi-party business processes with blockchain. In *Business Process Management: Blockchain and Central and Eastern Europe Forum - BPM 2019 Blockchain and CEE Forum, Vienna, Austria, September 1-6, 2019, Proceedings*, pages 55–70, 2019.

[102] Mirjam Minor, Ralph Bergmann, and Sebastian Görg. Case-based adaptation of workflows. *Inf. Syst.*, 40:142–152, 2014.

[103] Roman Mühlberger, Stefan Bachhofner, Claudio Di Ciccio, Luciano García-Bañuelos, and Orlenys López-Pintado. Extracting event logs for process mining from data stored on the blockchain. In *Business Process*

Management Workshops - BPM 2019 International Workshops, Vienna, Austria, September 1-6, 2019, Revised Selected Papers, pages 690–703, 2019.

[104] Robert Müller, Ulrike Greiner, and Erhard Rahm. $\mathrm{A_{gent}W_{ork}}$: a workflow system supporting rule-based workflow adaptation. *Data Knowl. Eng.*, 51(2):223–256, 2004.

[105] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[106] Aitor Murguzur, Karmele Intxausti, Aitor Urbieta, Salvador Trujillo, and Goiuria Sagardui. Process flexibility in service orchestration: A systematic literature review. *Int. J. Cooperative Inf. Syst.*, 23(3), 2014.

[107] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[108] Hiroaki Nakamura, Kohtaroh Miyamoto, and Michiharu Kudo. Inter-organizational business processes managed by blockchain. In *Web Information Systems Engineering - WISE 2018 - 19th International Conference, Dubai, United Arab Emirates, November 12-15, 2018, Proceedings, Part I*, pages 3–17, 2018.

[109] Anil Nigam and Nathan S. Caswell. Business artifacts: An approach to operational specification. *IBM Syst. J.*, 42(3):428–445, 2003.

[110] Alex Norta. Creation of smart-contracting collaborations for decentralized autonomous organizations. In *Perspectives in Business Informatics Research - 14th International Conference, BIR 2015, Tartu, Estonia, August 26-28, 2015, Proceedings*, pages 3–17, 2015.

[111] Alex Norta, Paul Grefen, and Nanjangud C. Narendra. A reference architecture for managing dynamic inter-organizational business processes. *Data Knowl. Eng.*, 91:52–89, 2014.

[112] Alex Norta, Lixin Ma, Yucong Duan, Addi Rull, Merit Kõlvart, and Kuldar Taveter. eContractual choreography-language properties towards cross- organizational business collaboration. *J. Internet Services and Applications*, 6(1):8:1–8:23, 2015.

[113] Oscar Novo. Blockchain meets iot: An architecture for scalable access management in iot. *IEEE Internet of Things Journal*, 5(2):1184–1195, 2018.

[114] Vanessa Tavares Nunes, Flávia Maria Santoro, Cláudia Maria Lima Werner, and Célia Ghedini Ralha. Real-time process adaptation: A context-aware replanning approach. *IEEE Trans. Systems, Man, and Cybernetics: Systems*, 48(1):99–118, 2018.

[115] Ivana Ognjanovic, Bardia Mohabbati, Dragan Gasevic, Ebrahim Bagheri, and Marko Boskovic. A metaheuristic approach for the configuration of business process families. In *2012 IEEE Ninth International Conference*

on Services Computing, Honolulu, HI, USA, June 24-29, 2012, pages 25–32, 2012.

[116] Aafaf Ouaddah, Anas Abou El Kalam, and Abdellah Ait Ouahman. Fairaccess: a new blockchain-based access control framework for the internet of things. *Security and Communication Networks*, 9(18):5943–5964, 2016.

[117] Chun Ouyang, Michael Adams, Arthur H.M. ter Hofstede, and Yang Yu. Towards the design of a scalable business process management system architecture in the cloud. In *Conceptual Modeling - 37th International Conference, ER 2018, Xi'an, China, October 22-25, 2018, Proceedings*, pages 334–348, 2018.

[118] Chun Ouyang, Marlon Dumas, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Jan Mendling. From business process models to process-oriented software systems. *ACM Trans. Softw. Eng. Methodol.*, 19(1):2:1–2:37, 2009.

[119] Lionel Palacin. Accelerate blockchain technology adoption with Bonita BPM and Chain Core, 2017. `https://www.bonitasoft.com/videos/secure-distributed-database-digital-assets-blockchain-and-bpm` – last accessed 2020-03-15.

[120] Cesare Pautasso and Gustavo 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*, pages 151–166, 2005.

[121] Maja Pesic, Helen Schonenberg, and Wil M. P. van der Aalst. DECLARE: full support for loosely-structured processes. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), 15-19 October 2007, Annapolis, Maryland, USA*, pages 287–300, 2007.

[122] Shaya Pourmirza, Remco Dijkman, and Paul Grefen. Switching parties in a collaboration at run-time. In *18th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2014, Ulm, Germany, September 1-5, 2014*, pages 136–141, 2014.

[123] Shaya Pourmirza, Sander Peters, Remco M. Dijkman, and Paul Grefen. A systematic literature review on the architecture of business process management systems. *Inf. Syst.*, 66:43–58, 2017.

[124] Shaya Pourmirza, Sander Peters, Remco M. Dijkman, and Paul Grefen. BPMS-RA: A novel reference architecture for business process management systems. *ACM Trans. Internet Techn.*, 19(1):13:1–13:23, 2019.

[125] Christoph Prybila, Stefan Schulte, Christoph Hochreiner, and Ingo Weber. Runtime verification for business processes utilizing the Bitcoin blockchain. *Future Generation Computer Systems (FGCS)*, 107:816–831, 2020.

[126] Luise Pufahl and Dimka Karastoyanova. Enhancing business process flex-

ibility by flexible batch processing. In *On the Move to Meaningful Internet Systems. OTM 2018 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22-26, 2018, Proceedings, Part I*, pages 426–444, 2018.

[127] Heorhi Raik, Antonio Bucchiarone, Nawaz Khurshid, Annapaola Marconi, and Marco Pistore. Astro-captevo: Dynamic context-aware adaptation for service-based systems. In *Eighth IEEE World Congress on Services, SERVICES 2012, Honolulu, HI, USA, June 24-29, 2012*, pages 385–392. IEEE Computer Society, 2012.

[128] Manfred Reichert and Barbara Weber. *Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies*. Springer, 2012.

[129] Gartner Press Release. Gartner survey reveals the scarcity of current blockchain deployments, 2018. `https://www.gartner.com/en/newsroom/press-releases/2018-05-03-gartner-survey-reveals-the-scarcity-of-current-blockchain-developments` – last accessed 2020-03-15.

[130] Olivier Rikken. BPM and blockchain, miles apart or closer than you think?, 2015. `https://www.bpmleader.com/2015/11/17/bpm-blockchain-miles-apart-closer-think/` – last accessed 2020-03-15.

[131] Paul Rimba, An Binh Tran, Ingo Weber, Mark Staples, Alexander Ponomarev, and Xiwei Xu. Comparing blockchain and cloud services for business process execution. In *2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017*, pages 257–260, 2017.

[132] Philip Robinson, Florian Kerschbaum, and Andreas 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*, pages 297–309, 2006.

[133] Marcello La Rosa, Wil M. P. van der Aalst, Marlon Dumas, and Fredrik Milani. Business process variability modeling: A survey. *ACM Comput. Surv.*, 50(1):2:1–2:45, 2017.

[134] Stephan Roser, Jörg P. Müller, and Bernhard Bauer. An evaluation and decision method for ICT architectures for cross-organizational business process coordination. *Inf. Syst. E-Business Management*, 9(1):51–88, 2011.

[135] Sara Rouhani and Ralph Deters. Blockchain based access control systems: State of the art and challenges. In *2019 IEEE/WIC/ACM International Conference on Web Intelligence, WI 2019, Thessaloniki, Greece, October 14-17, 2019*, pages 423–428, 2019.

[136] Nick Russell, Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, and David 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*, pages 216–232, 2005.

[137] Shazia W. Sadiq, Wasim Sadiq, and Maria 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*, pages 513–526, 2001.

[138] Shazia Wasim Sadiq, Guido Governatori, and Kioumars Namiri. Modeling control objectives for business process compliance. In *Proceedings of the 5th International Conference on Business Process Management (BPM)*, volume 4714 of *Lecture Notes in Computer Science*, pages 149–164, 2007.

[139] Shazia Wasim Sadiq, Maria E. Orlowska, and Wasim Sadiq. Specification and validation of process constraints for flexible workflows. *Inf. Syst.*, 30(5):349–378, 2005.

[140] Helen Schonenberg, Ronny Mans, Nick Russell, Nataliya Mulyar, and Wil 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*, pages 16–30, 2008.

[141] Helen Schonenberg, Ronny Mans, Nick Russell, Nataliya Mulyar, and Wil 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*, pages 81–84, 2008.

[142] Emmanuel Boateng Sifah, Qi Xia, Kwame Opuni-Boachie Obour Agyekum, Sandro Amofa, Jianbin Gao, Ruidong Chen, Hu Xia, James C. Gee, Xiaojiang Du, and Mohsen Guizani. Chain-based big data access control infrastructure. *The Journal of Supercomputing*, 74(10):4945–4964, 2018.

[143] Renuka Sindhgatta, Aditya K. Ghose, and Hoa Khanh Dam. Context-aware analysis of past process executions to aid resource allocation decisions. In *Advanced Information Systems Engineering - 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings*, pages 575–589, 2016.

[144] Wei Song and Hans-Arno Jacobsen. Static and dynamic process change. *IEEE Trans. Serv. Comput.*, 11(1):215–231, 2018.

[145] Mirko Sonntag and Dimka Karastoyanova. Concurrent workflow evolution. *ECEASST*, 37, 2011.

[146] Mark Staples, Shiping Chen, Sara Falamaki, Alex Ponomarev, Paul Rimba, An Binh Tran, Ingo Weber, Xiwei Xu, and John Zhu. Risks and opportunities for systems using blockchain and smart contracts. Technical report, Data61(CSIRO), Sydney, 2017.

[147] Christian Sturm, Jonas Scalanczi, Stefan Schönig, and Stefan Jablonski. A blockchain-based and resource-aware process execution engine. *Future Gener. Comput. Syst.*, 100:19–34, 2019.

[148] Christian Sturm, Jonas Szalanczi, Stefan Schönig, and Stefan Jablonski. A lean architecture for blockchain based decentralized process execution. In *Business Process Management Workshops - BPM 2018 International Workshops, Sydney, NSW, Australia, September 9-14, 2018, Revised Papers*, pages 361–373, 2018.

[149] Simon Tragatschnig, Srdjan Stevanetic, and Uwe Zdun. Supporting the evolution of event-driven service-oriented architectures using change patterns. *Inf. Softw. Technol.*, 100:133–146, 2018.

[150] An Binh Tran, Qinghua Lu, and Ingo 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.*, pages 56–60, 2018.

[151] Ching-Hong Tsai, Kuo-Chan Huang, Feng-Jian Wang, and Chun-Hao Chen. A distributed server architecture supporting dynamic resource provisioning for bpm-oriented workflow management systems. *Journal of Systems and Software*, 83(8):1538–1552, 2010.

[152] UK Government Chief Scientific Adviser. Distributed ledger technology: Beyond block chain. Technical Report 19, UK Government Office of Science, 2016.

[153] Nick R.T.P. van Beest, Eirini Kaldeli, Pavel Bulanov, Johan C. Wortmann, and Alexander Lazovik. Automated runtime repair of business processes. *Inf. Syst.*, 39:45–79, 2014.

[154] Wil M. P. van der Aalst. Business process management: a comprehensive survey. *ISRN Software Engineering*, 2013, 2013.

[155] Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, and Mathias Weske. Business process management: A survey. In *Business Process Management, International Conference, BPM 2003, Eindhoven, The Netherlands, June 26-27, 2003, Proceedings*, pages 1–12, 2003.

[156] Wattana Viriyasitavat and Danupol Hoonsopon. Blockchain characteristics and consensus in modern business processes. *Journal of Industrial Information Integration*, 13:32–39, 2019.

[157] Jacques Wainer, Akhil Kumar, and Paulo Barthelmess. DW-RBAC: A formal security model of delegation and revocation in workflow systems. *Inf. Syst.*, 32(3):365–384, 2007.

[158] Jiacun Wang, Yi Deng, and Gang Xu. Reachability analysis of real-time

systems using time petri nets. *IEEE Trans. Systems, Man, and Cybernetics, Part B*, 30(5):725–736, 2000.

[159] Shangping Wang, Yinglong Zhang, and Yaling Zhang. A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems. *IEEE Access*, 6:38437–38450, 2018.

[160] Barbara Weber, Shazia Wasim Sadiq, and Manfred Reichert. Beyond rigidity - dynamic process lifecycle support. *Comput. Sci. Res. Dev.*, 23(2):47–65, 2009.

[161] Ingo Weber, Jochen Haller, and Jutta A. Mülle. Automated derivation of executable business processes from choreographies in virtual organisations. *Int. J. Bus. Process. Integr. Manag.*, 3(2):85–95, 2008.

[162] Ingo Weber, Qinghua Lu, An Binh Tran, Amit Deshmukh, Marek Górski, and Markus Strazds. A platform architecture for multi-tenant blockchain-based systems. In *IEEE International Conference on Software Architecture, ICSA 2019, Hamburg, Germany, March 25-29, 2019*, pages 101–110, 2019.

[163] Ingo Weber, Xiwei Xu, Régis Riveret, Guido Governatori, Alexander Ponomarev, and Jan 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*, pages 329–347, 2016.

[164] Andreas Weiß, Santiago Gómez Sáez, Michael Hahn, and Dimka Karastoyanova. Approach and refinement strategies for flexible choreography enactment. In *On the Move to Meaningful Internet Systems: OTM 2014 Conferences - Confederated International Conferences: CoopIS, and ODBASE 2014, Amantea, Italy, October 27-31, 2014, Proceedings*, pages 93–111, 2014.

[165] Mathias Weske. *Business Process Management - Concepts, Languages, Architectures, Third Edition*. Springer, 2019.

[166] Dirk Wodtke, Jeanine Weißenfels, Gerhard Weikum, and Angelika Kotz Dittrich. The mentor project: Steps toward enterprise-wide workflow management. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana, USA*, pages 556–565, 1996.

[167] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[168] Karl Wüst and Arthur Gervais. Do you need a blockchain? In *Crypto Valley Conference on Blockchain Technology, CVCBT 2018, Zug, Switzerland, June 20-22, 2018*, pages 45–54, 2018.

[169] Qi Xia, Emmanuel Boateng Sifah, Kwame Omono Asamoah, Jianbin Gao, Xiaojiang Du, and Mohsen Guizani. Medshare: Trust-less medical data

sharing among cloud service providers via blockchain. *IEEE Access*, 5:14757–14767, 2017.

[170] Zan Xiao, Donggang Cao, Chao You, and Hong Mei. Towards a constraint-based framework for dynamic business process adaptation. In *IEEE International Conference on Services Computing, SCC 2011, Washington, DC, USA, 4-9 July, 2011*, pages 685–692, 2011.

[171] Wei Xu, Jianwen Su, Zhimin Yan, Jian Yang, and Liang Zhang. An artifact-centric approach to dynamic modification of workflow execution. In *On the Move to Meaningful Internet Systems: OTM 2011 - Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2011, Hersonissos, Crete, Greece, October 17-21, 2011, Proceedings, Part I*, volume 7044 of *Lecture Notes in Computer Science*, pages 256–273. Springer, 2011.

[172] Xiwei Xu, Cesare Pautasso, Liming Zhu, Vincent Gramoli, Alexander Ponomarev, An Binh Tran, and Shiping Chen. The blockchain as a software connector. In *13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016, Venice, Italy, April 5-8, 2016*, pages 182–191, 2016.

[173] Xiwei Xu, Ingo Weber, and Mark Staples. *Architecture for Blockchain Applications*. Springer, 2019.

[174] Xiwei Xu, Ingo Weber, Mark Staples, Liming Zhu, Jan Bosch, Len Bass, Cesare Pautasso, and Paul Rimba. A taxonomy of blockchain-based systems for architecture design. In *2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017*, pages 243–252, 2017.

[175] Rajitha Yasaweerasinghelage, Mark Staples, and Ingo Weber. Predicting latency of blockchain-based systems using architectural modelling and simulation. In *2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017*, pages 253–256, 2017.

[176] Matteo Zavatteri, Carlo Combi, Roberto Posenato, and Luca 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*, pages 235–251, 2017.

[177] Christian Zeyen, Lukas Malburg, and Ralph Bergmann. Adaptation of scientific workflows by means of process-oriented case-based reasoning. In *Case-Based Reasoning Research and Development - 27th International Conference, ICCBR 2019, Otzenhausen, Germany, September 8-12, 2019, Proceedings*, pages 388–403, 2019.

[178] Daoye Zhang, Dahai Cao, Lijie Wen, and Jianmin Wang. An efficient approach for supporting dynamic evolutionary change of adaptive workflow. In *Progress in WWW Research and Development, 10th Asia-Pacific Web Conference, APWeb 2008, Shenyang, China, April 26-28, 2008. Proceedings*, pages 684–695, 2008.

[179] Liang-Jie Zhang and Qun Zhou. CCOA: cloud computing open architecture. In *IEEE International Conference on Web Services, ICWS 2009, Los Angeles, CA, USA, 6-10 July 2009*, pages 607–616, 2009.

[180] Xiaohui Zhao and Chengfei Liu. Version management for business process schema evolution. *Inf. Syst.*, 38(8):1046–1069, 2013.

[181] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. Blockchain challenges and opportunities: a survey. *IJWGS*, 14(4):352–375, 2018.

Appendix A. CODE REPOSITORIES

The source code of Caterpillar can be downloaded under the BSD 3-clause “New" or “Revised" License from `https://github.com/orlenyslp/Caterpillar`. The repository splits the contributions described in this thesis into four different versions of CATERPILLAR.

- V1.0 provides an initial implementation presented in the demo paper titled “*Caterpillar: A Blockchain-Based Business Process Management System*” [89].
- V2.0 enhances the architecture implemented in V1.0, in correspondence to the contribution presented in the Chapter 4 of this thesis, and published in the paper titled “CATERPILLAR*: A Business Process Execution Engine on the Ethereum Blockchain*”[90].
- V2.1 extends the prototype with the dynamic binding access control, and the agreement policies aligned with the contribution described in the Chapter 6 of this thesis, and the paper titled “*Dynamic Role Binding in Blockchain-Based Collaborative Business Processes*” [87].
- V3.0 includes a new engine supporting the architecture to execute interpreted processes corresponding to contribution described in the Chapter 5 of this thesis, and the paper titled “*Interpreted Execution of Business Process Models on Blockchain*” [88].
- CATERPILLAR-FULL contains the full implementation of the CATERPILLAR SYSTEM, including the compilation-based and interpretation-based engines, and the controlled flexibility mechanisms. This implementation integrates all the results presented in the thesis into a single system and constitutes the last version of the code.

CATERPILLAR’s code distribution contains two folders: (i) folder *caterpillar_core*, which includes the implementation of the core components (engine, compilation-interpreter tools, event monitor and work item manager) and (ii) *execution_panel* (implementation of the Execution Panel module).

The repository contains all instructions needed to install the required dependencies and running the sample process models.

ACKNOWLEDGEMENT

I am very grateful to my supervisors Marlon Dumas and Luciano García Bañuelos for the guidance, the support and feedback throughout these years. I am thankful for the constant opportunities for professional development they gave me.

I want to thank Ingo Weber for his time while hosting me in his research group at Data61, and for his help to refine and execute many ideas as co-author of several research papers.

I want to thank all my friends for all the memories accumulated across the path to complete the PhD. To all my colleges and friends at the University of Tartu in Estonia, and Data61 in Australia for their support. I want to give special thanks to my friends Dayron, Lídice and Jose, who made it possible that I could move to Estonia four years ago. To Arlete and Lucas, who shared their home in Sydney and made me part of the family for six months. To Irene and Justin who helped me with the Estonian translations in this thesis. Finally, I would like to thank my mother, father and brother for being always with me, breaking all the physical distances that may separate us.

I would also like to acknowledge the Estonian Research Council, the Doctoral School of Information and Communication Technology (IKTDK), and the European Regional Development Fund for funding my studies.

SUMMARY IN ESTONIAN

Koostööäriprotsesside läbiviimine plokiahelal: Caterpillari süsteem

Tänapäeval peavad organisatsioonid tegema omavahel koostööd, et kasutada ära üksteise täiendavaid võimekusi ning seeläbi pakkuda oma klientidele parimaid tooteid ja teenuseid. Selleks peavad organisatsioonid juhtima äriprotsesse, mis ületavad nende organisatsioonilisi piire. Selliseid protsesse nimetatakse koostööäriprotsessideks.

Üks peamisi takistusi koostööäriprotsesside elluviimisel on osapooltevahelise usalduse puudumine. Viimase kümnendi jooksul on üldise lahendusena välja kujunenud plokiahela tehnoloogia, mis võimaldab pooltel teha koostööd vastastikuse usalduse puudumisel. Plokiahela tehnoloogia võimaldab osapooltel ülal pidada muutmatut hajutatud transaktsioonide registrit ja juurutada programme (nn "nutilepinguid"), mis käivitatakse, kui teatud transaktsioonid aset leiavad. Neid funktsioone saab kasutada oluliste ehitusplokkidena koostööprotsesside läbiviimisel, kui osapoolte vahel puudub vastastikune usaldus. Paraku on aga äriprotsesside läbiviimine selliseid madala taseme plokiahela elemente kasutades tülikas ja veaohtlik. Seevastu juba väljakujunenud äriprotsesside juhtimissüsteemid (BUSINESS PROCESS MANAGEMENT SYSTEM – BPMS), näiteks BPMN (BUSINESS PROCESS MODEL AND NOTATION) standardil põhinevad süsteemid, pakuvad käepäraseid abstraheeringuid protsessidele orienteeritud rakenduste kiireks arendamiseks.

Käesolev doktoritöö käsitleb koostööäriprotsesside automatiseeritud läbiviimist plokiahela tehnoloogiat kasutades, kombineerides traditsiooniliste BPMS-ide arendusvõimalused plokiahelast tuleneva suurendatud usaldusega. Samuti käsitleb antud doktoritöö küsimust, kuidas pakkuda tuge olukordades, kus uued osapooled võivad jooksvalt protsessiga liituda, mistõttu on vajalik tagada paindlikkus äriprotsessi marsruutimisloogika muutmise osas, tagades samal ajal, et protsessi läbiviimine jätkaks oma põhispetsifikatsiooni kohast kulgu.

Antud doktoritöö sõnastab põhimõtted ja nõuded koostööäriprotsesside läbiviimiseks plokiahelal. Seejärel pakub doktoritöö välja ja hindab arhitektuurilisi lähenemisviise ning protsesside modelleerimise kontseptsioone nende põhimõtete ja nõuete täitmiseks. Arhitektuurilised lähenemisviisid ja modelleerimise kontseptsioonid on koondatud uudsesse plokiahelal põhinevasse BPMS-i nimega CATERPILLAR. Nagu iga protsessi läbiviimise mootor, toetab CATERPILLAR protsessimudeli instantside loomist ning võimaldab kasutajatel jälgida protsessiinstantside olekut ja täita nendega seonduvaid ülesandeid. CATERPILLAR eristub teistest BPMS-idest selle poolest, et iga protsessiinstantsi olek säilitatakse (Ethereum) plokiahelal ja töövoo marsruutimist teostatakse läbi nutilepingute.

CATERPILLAR-i süsteem toetab kahte lähenemist plokiahelal põhinevate protsesside rakendamiseks, läbiviimiseks ja seireks: kompileeritud lähenemine ja tõl-

gendatatud lähenemine. Kompileeritud lähenemine tugineb Solidity programmeerimiskeele kompilaatorile BPMN modelleerimisnotatsioonist. Kompilaator toetab ulatuslikku hulka BPMN-konstruktsioone, sealhulgas alamprotsesse, mitmeinstantsilisi tegevusi ja sündmuste käitlejaid. See toetab ka protsesse, mis on täiustatud andmepiirangutega, mis suunavad protsessi läbiviimist.

Kompileeritud lähenemine kasutab täiel määral plokiahela platvormide muutmatuse omadusi – pärast protsessi juurutamist ei saa selle äriloogikat enam muuta. Teisalt ei ole see lähenemisviis sobiv dünaamilise koostöö juhtude puhul, kus paindlikkus on nõudeks. Et käsitleda dünaamilise koostöö juhte, toetab CATERPILLAR ka tõlgendatud lähenemist äriprotsesside läbiviimiseks. See tugineb BPMN mudelite interpretaatoril, põhineb dünaamilistel andmestruktuuridel, mis on manustatud äriprotsessi läbiviimise süsteemi modulaarse mitmekihilise arhitektuuriga, toetades protsessiinstantside loomist, läbiviimist, seiret ja dünaamilist uuendamist. Tõhususe eesmärgil tugineb interpretaator kompaktsetel bitmapil põhinevatel protsessimudelite kodeeringutel. Eksperimentide kohaselt saavutab välja pakutud tõlgendatud lähenemisviis olemasolevate kompileeritud lahendustega võrreldes samaväärsed või väiksemad kulud.

Kuigi paindlikkus on soovitav omadus, tuleb seda piirata, et vältida olukorda, kus mõni osapool juhib protsessi teisi kahjustaval viisil. Selle probleemi lahendamiseks pakub käesolev doktoritöö välja kaks mudelit kontrollitud paindlikkuse võimaldamiseks koostööäriprotsessides. Esiteks pakub doktoritöö välja mudeli osalejate dünaamiliseks sidumiseks rollidega koostööäriprotsessides ning sellega seotud siduvate eeskirjade spetsifikatsioonikeele. Välja pakutud keel on varustatud Petri võrkude semantikaga, võimaldades seega eeskirjade järjepidevuse kontrollimist. Teiseks pakub doktoritöö välja mudeli juhtimisvoo konsensuspõhise paindlikkuse jaoks, mille kohaselt saavad protsessis osalejad kollektiivselt kokku leppida, kuidas äriprotsessi juhtimisvoo kokkulepete eeskirjade poolt määratletud piirides juhtida. Antud doktoritöö pakub välja ka lähenemise eeskirjade spetsifikatsioonide kompileerimiseks jõustatavateks nutilepinguteks. Eksperimendid näitavad, et eeskirja jõustamise kulud suurenevad lineaarselt rollide, juhtimisvoo elementide ja eeskirja piirangute arvude suhtes.

CURRICULUM VITAE

Personal data

Name:	Orlenys López-Pintado
Date of Birth:	18.09.1985
Citizenship:	Cuban

Education

2016–2020	University of Tartu, Faculty of Science and Technology, doctoral studies, specialty: Computer Science.
2012–2015	University of Havana, Faculty of Mathematics and Computer Science, master's studies, specialty: Mathematics Science.
2004–2010	University of Havana, Faculty of Mathematics and Computer Science, bachelor's studies, specialty: Computer Science.

Employment

2019 – 2020	University of Tartu, Institute of Computer Science, Junior Research Fellow of Information Systems
2017 – 2019	University of Tartu, Institute of Computer Science, Teaching Assistant
2015 – 2016	Agricultural University of Havana, Department of Informatics, Assistant Professor
2012 – 2015	Agricultural University of Havana, Department of Informatics, Instructor Professor
2010 – 2012	Agricultural University of Havana, Department of Informatics, Instructor Professor (In Training)

Scientific work

Main fields of interest:

- blockchain
- information systems
- business process management

ELULOOKIRJELDUS

Isikuandmed

Nimi: Orlenys López-Pintado
Sünniaeg: 18.09.1985
Kodakondsus: Kuuba

Haridus

2016–2020 Tartu Ülikool, loodus- ja täppisteaduste valdkond, doktoriõpe, eriala: informaatika.

2012–2015 Havanna Ülikool, matemaatika- ja arvutiteaduskond, magistriõpe, eriala: matemaatika.

2004–2010 Havanna Ülikool, matemaatika- ja arvutiteaduskond, bakalaureuseõpe, eriala: informaatika.

Teenistuskäik

2019 – 2020 Tartu Ülikool, arvutiteaduse instituut, infosüsteemide nooremteadur

2017 – 2019 Tartu Ülikool, arvutiteaduse instituut, praktikumijuhendaja

2015 – 2016 Havanna Põllumajandusülikool, informaatika rühm, dotsent

2012 – 2015 Havanna Põllumajandusülikool, informaatika rühm, juhendaja professor

2010 – 2012 Havanna Põllumajandusülikool, informaatika rühm, juhendaja professor (treeningus)

Teadustegevus

Peamised uurimisvaldkonnad:

- plokiahel
- infosüsteemid
- äriprotsesside juhtimine

LIST OF ORIGINAL PUBLICATIONS

Publications in the scope of the thesis

I Orlenys López-Pintado, Luciano García-Bañuelos, Marlon Dumas, and Ingo Weber. Caterpillar: A blockchain-based business process management system. In *Proceedings of the Demo Track and Dissertation Award of the 15th International Conference on Business Process Management (BPM 2017), Barcelona, Spain, September 13, 2017*, 2017

II Orlenys López-Pintado. Business process execution on blockchain. In *Proceedings of the Doctoral Consortium at the 30th International Conference on Advanced Information Systems Engineering (CAiSE 2018), Tallinn, Estonia, June 11-15, 2018*, volume 2114 of *CEUR Workshop Proceedings*, pages 10–18, 2018

III Orlenys López-Pintado, Luciano García-Bañuelos, Marlon Dumas, Ingo Weber, and Alexander Ponomarev. Caterpillar: A business process execution engine on the ethereum blockchain. *Softw., Pract. Exper.*, 49(7):1162–1193, 2019

IV Orlenys López-Pintado, Marlon Dumas, Luciano García-Bañuelos, and Ingo Weber. Interpreted execution of business process models on blockchain. In *23rd IEEE International Enterprise Distributed Object Computing Conference, EDOC 2019, Paris, France, October 28-31, 2019*, pages 206–215, 2019

V Orlenys López-Pintado, Marlon Dumas, Luciano García-Bañuelos, and Ingo 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*, pages 399–414, 2019

Publications out of the scope of the thesis

I Claudio Di Ciccio, Alessio Cecconi, Marlon Dumas, Luciano García-Bañuelos, Orlenys López-Pintado, Qinghua Lu, Jan Mendling, Alexander Ponomarev, An Binh Tran, and Ingo Weber. Blockchain support for collaborative business processes. *Informatik Spektrum*, 42(3):182–190, 2019

II Roman Mühlberger, Stefan Bachhofner, Claudio Di Ciccio, Luciano García-Bañuelos, and Orlenys López-Pintado. Extracting event logs for process mining from data stored on the blockchain. In *Business Process Management Workshops - BPM 2019 International Workshops, Vienna, Austria, September 1-6, 2019, Revised Selected Papers*, pages 690–703, 2019

DISSERTATIONES INFORMATICAE PREVIOUSLY PUBLISHED IN DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

19. **Helger Lipmaa.** Secure and efficient time-stamping systems. Tartu, 1999, 56 p.
22. **Kaili Müürisep.** Eesti keele arvutigrammatika: süntaks. Tartu, 2000, 107 lk.
23. **Varmo Vene.** Categorical programming with inductive and coinductive types. Tartu, 2000, 116 p.
24. **Olga Sokratova.** Ω-rings, their flat and projective acts with some applications. Tartu, 2000, 120 p.
27. **Tiina Puolakainen.** Eesti keele arvutigrammatika: morfoloogiline ühestamine. Tartu, 2001, 138 lk.
29. **Jan Villemson.** Size-efficient interval time stamps. Tartu, 2002, 82 p.
45. **Kristo Heero.** Path planning and learning strategies for mobile robots in dynamic partially unknown environments. Tartu 2006, 123 p.
49. **Härmel Nestra.** Iteratively defined transfinite trace semantics and program slicing with respect to them. Tartu 2006, 116 p.
53. **Marina Issakova.** Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment. Tartu 2007, 170 p.
55. **Kaarel Kaljurand.** Attempto controlled English as a Semantic Web language. Tartu 2007, 162 p.
56. **Mart Anton.** Mechanical modeling of IPMC actuators at large deformations. Tartu 2008, 123 p.
59. **Reimo Palm.** Numerical Comparison of Regularization Algorithms for Solving Ill-Posed Problems. Tartu 2010, 105 p.
61. **Jüri Reimand.** Functional analysis of gene lists, networks and regulatory systems. Tartu 2010, 153 p.
62. **Ahti Peder.** Superpositional Graphs and Finding the Description of Structure by Counting Method. Tartu 2010, 87 p.
64. **Vesal Vojdani.** Static Data Race Analysis of Heap-Manipulating C Programs. Tartu 2010, 137 p.
66. **Mark Fišel.** Optimizing Statistical Machine Translation via Input Modification. Tartu 2011, 104 p.
67. **Margus Niitsoo**. Black-box Oracle Separation Techniques with Applications in Time-stamping. Tartu 2011, 174 p.
71. **Siim Karus.** Maintainability of XML Transformations. Tartu 2011, 142 p.
72. **Margus Treumuth.** A Framework for Asynchronous Dialogue Systems: Concepts, Issues and Design Aspects. Tartu 2011, 95 p.
73. **Dmitri Lepp.** Solving simplification problems in the domain of exponents, monomials and polynomials in interactive learning environment T-algebra. Tartu 2011, 202 p.

74. **Meelis Kull.** Statistical enrichment analysis in algorithms for studying gene regulation. Tartu 2011, 151 p.
77. **Bingsheng Zhang.** Efficient cryptographic protocols for secure and private remote databases. Tartu 2011, 206 p.
78. **Reina Uba.** Merging business process models. Tartu 2011, 166 p.
79. **Uuno Puus.** Structural performance as a success factor in software development projects – Estonian experience. Tartu 2012, 106 p.
81. **Georg Singer.** Web search engines and complex information needs. Tartu 2012, 218 p.
83. **Dan Bogdanov.** Sharemind: programmable secure computations with practical applications. Tartu 2013, 191 p.
84. **Jevgeni Kabanov.** Towards a more productive Java EE ecosystem. Tartu 2013, 151 p.
87. **Margus Freudenthal.** Simpl: A toolkit for Domain-Specific Language development in enterprise information systems. Tartu, 2013, 151 p.
90. **Raivo Kolde.** Methods for re-using public gene expression data. Tartu, 2014, 121 p.
91. **Vladimir Šor.** Statistical Approach for Memory Leak Detection in Java Applications. Tartu, 2014, 155 p.
92. **Naved Ahmed.** Deriving Security Requirements from Business Process Models. Tartu, 2014, 171 p.
94. **Liina Kamm.** Privacy-preserving statistical analysis using secure multi-party computation. Tartu, 2015, 201 p.
100. **Abel Armas Cervantes.** Diagnosing Behavioral Differences between Business Process Models. Tartu, 2015, 193 p.
101. **Fredrik Milani.** On Sub-Processes, Process Variation and their Interplay: An Integrated Divide-and-Conquer Method for Modeling Business Processes with Variation. Tartu, 2015, 164 p.
102. **Huber Raul Flores Macario.** Service-Oriented and Evidence-aware Mobile Cloud Computing. Tartu, 2015, 163 p.
103. **Tauno Metsalu.** Statistical analysis of multivariate data in bioinformatics. Tartu, 2016, 197 p.
104. **Riivo Talviste.** Applying Secure Multi-party Computation in Practice. Tartu, 2016, 144 p.
108. **Siim Orasmaa.** Explorations of the Problem of Broad-coverage and General Domain Event Analysis: The Estonian Experience. Tartu, 2016, 186 p.
109. **Prastudy Mungkas Fauzi.** Efficient Non-interactive Zero-knowledge Protocols in the CRS Model. Tartu, 2017, 193 p.
110. **Pelle Jakovits.** Adapting Scientific Computing Algorithms to Distributed Computing Frameworks. Tartu, 2017, 168 p.
111. **Anna Leontjeva.** Using Generative Models to Combine Static and Sequential Features for Classification. Tartu, 2017, 167 p.
112. **Mozhgan Pourmoradnasseri.** Some Problems Related to Extensions of Polytopes. Tartu, 2017, 168 p.

113. **Jaak Randmets.** Programming Languages for Secure Multi-party Computation Application Development. Tartu, 2017, 172 p.
114. **Alisa Pankova.** Efficient Multiparty Computation Secure against Covert and Active Adversaries. Tartu, 2017, 316 p.
116. **Toomas Saarsen.** On the Structure and Use of Process Models and Their Interplay. Tartu, 2017, 123 p.
121. **Kristjan Korjus.** Analyzing EEG Data and Improving Data Partitioning for Machine Learning Algorithms. Tartu, 2017, 106 p.
122. **Eno Tõnisson.** Differences between Expected Answers and the Answers Offered by Computer Algebra Systems to School Mathematics Equations. Tartu, 2017, 195 p.

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

1. **Abdullah Makkeh**. Applications of Optimization in Some Complex Systems. Tartu 2018, 179 p.
2. **Riivo Kikas**. Analysis of Issue and Dependency Management in Open-Source Software Projects. Tartu 2018, 115 p.
3. **Ehsan Ebrahimi**. Post-Quantum Security in the Presence of Superposition Queries. Tartu 2018, 200 p.
4. **Ilya Verenich**. Explainable Predictive Monitoring of Temporal Measures of Business Processes. Tartu 2019, 151 p.
5. **Yauhen Yakimenka**. Failure Structures of Message-Passing Algorithms in Erasure Decoding and Compressed Sensing. Tartu 2019, 134 p.
6. **Irene Teinemaa**. Predictive and Prescriptive Monitoring of Business Process Outcomes. Tartu 2019, 196 p.
7. **Mohan Liyanage.** A Framework for Mobile Web of Things. Tartu 2019, 131 p.
8. **Toomas Krips.** Improving performance of secure real-number operations. Tartu 2019, 146 p.
9. **Vijayachitra Modhukur.** Profiling of DNA methylation patterns as biomarkers of human disease. Tartu 2019, 134 p.
10. **Elena Sügis.** Integration Methods for Heterogeneous Biological Data. Tartu 2019, 250 p.
11. **Tõnis Tasa.** Bioinformatics Approaches in Personalised Pharmacotherapy. Tartu 2019, 150 p.
12. **Sulev Reisberg.** Developing Computational Solutions for Personalized Medicine. Tartu 2019, 126 p.
13. **Huishi Yin.** Using a Kano-like Model to Facilitate Open Innovation in Requirements Engineering. Tartu 2019, 129 p.
14. **Faiz Ali Shah.** Extracting Information from App Reviews to Facilitate Software Development Activities. Tartu 2020, 149 p.
15. **Adriano Augusto**. Accurate and Efficient Discovery of Process Models from Event Logs. Tartu 2020, 194 p.
16. **Karim Baghery.** Reducing Trust and Improving Security in zk-SNARKs and Commitments. Tartu 2020, 245 p.
17. **Behzad Abdolmaleki.** On Succinct Non-Interactive Zero-Knowledge Protocols Under Weaker Trust Assumptions. Tartu 2020, 209 p.
18. **Janno Siim.** Non-Interactive Shuffle Arguments. Tartu 2020, 154 p.
19. **Ilya Kuzovkin.** Understanding Information Processing in Human Brain by Interpreting Machine Learning Models. Tartu 2020, 149 p.