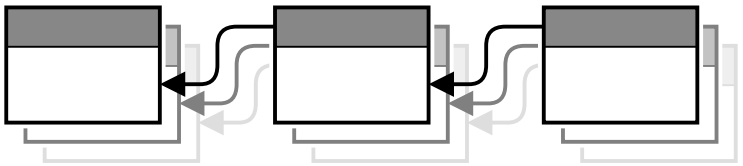
Enforceability Aspects of — Smart Contracts — on Blockchain Networks



Jan Ladleif

Enforceability Aspects of Smart Contracts on Blockchain Networks

Jan Ladleif

Business Process Technology Group

Hasso Plattner Institute
Digital Engineering Faculty
University of Potsdam

Potsdam, Germany

— Dissertation —

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
Dr. rer. nat.

Date of Defense: September 16th, 2021

May 2021

Unless otherwise indicated, this work is licensed under a Creative Commons License
Attribution 4.0 International.
This does not apply to quoted content and works based on other permissions.
To view a copy of this license visit:
https://creativecommons.org/licenses/by/4.0

Supervisor: Prof. Dr. Mathias Weske, University of Potsdam
Reviewers: Prof. Dr. Cesare Pautasso, University of Lugano, *and*
Prof. Dr. Jan Mendling, Humboldt University of Berlin

Published online on the
Publication Server of the University of Potsdam:
https://doi.org/10.25932/publishup-51908
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-519088

Abstract

Smart contracts promise to reform the legal domain by automating clerical and procedural work, and minimizing the risk of fraud and manipulation. Their core idea is to draft contract documents in a way which allows machines to process them, to grasp the operational and non-operational parts of the underlying legal agreements, and to use tamper-proof code execution alongside established judicial systems to enforce their terms.

The implementation of smart contracts has been largely limited by the lack of an adequate technological foundation which does not place an undue amount of trust in any contract party or external entity. Only recently did the emergence of Decentralized Applications (DApps) change this: Stored and executed via transactions on novel distributed ledger and blockchain networks, powered by complex integrity and consensus protocols, DApps grant secure computation and immutable data storage while at the same time eliminating virtually all assumptions of trust.

However, research on how to effectively capture, deploy, and most of all enforce smart contracts with DApps in mind is still in its infancy. Starting from the initial expression of a smart contract's intent and logic, to the operation of concrete instances in practical environments, to the limits of automatic enforcement—many challenges remain to be solved before a widespread use and acceptance of smart contracts can be achieved.

This thesis proposes a model-driven smart contract management approach to tackle some of these issues. A metamodel and semantics of smart contracts are presented, containing concepts such as legal relations, autonomous and non-autonomous actions, and their interplay. Guided by the metamodel, the notion and a system architecture of a Smart Contract Management System (SCMS) is introduced, which facilitates smart contracts in all phases of their lifecycle. Relying on DApps in heterogeneous multi-chain environments, the SCMS approach is evaluated by a proof-of-concept implementation showing both its feasibility and its limitations.

Further, two specific enforceability issues are explored in detail: The performance of fully autonomous tamper-proof behavior with external off-chain dependencies and the evaluation of temporal constraints within DApps, both of which are essential for smart contracts but challenging to support in the restricted transaction-driven and closed environment of blockchain networks. Various strategies of implementing or emulating these capabilities, which are ultimately applicable to all kinds of DApp projects independent of smart contracts, are presented and evaluated.

Publications

Some ideas and figures found in this thesis have previously appeared in the following publications:

1. Jan Ladleif and Mathias Weske. Time in blockchain-based process execution. In *24th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2020, Eindhoven, The Netherlands, October 5-8, 2020*, pages 217–226. IEEE, 2020. doi: 10.1109/EDOC49727.2020.00034

2. Jan Ladleif, Ingo Weber, and Mathias Weske. External data monitoring using oracles in blockchain-based process execution. In Aleksandre Asatiani et al., editors, *Business Process Management: Blockchain and Robotic Process Automation Forum, BPM 2020*, volume 393 of *Lecture Notes in Business Information Processing*, pages 67–81. Springer, Cham, 2020. doi: 10.1007/978-3-030-58779-6_5

3. Jan Ladleif, Christian Friedow, and Mathias Weske. An architecture for multi-chain business process choreographies. In Witold Abramowicz and Gary Klein, editors, *Business Information Systems*, volume 389 of *Lecture Notes in Business Information Processing*, pages 184–196. Springer, Cham, 2020. doi: 10.1007/978-3-030-53337-3_14

4. Jan Ladleif and Mathias Weske. A unifying model of legal smart contracts. In Alberto H. F. Laender et al., editors, *Conceptual Modeling*, volume 11788 of *Lecture Notes in Computer Science*, pages 323–337. Springer, Cham, 2019. doi: 10.1007/978-3-030-33223-5_27

5. Jan Ladleif, Anton von Weltzien, and Mathias Weske. chor-js: A modeling framework for BPMN 2.0 choreography diagrams. In José Ignacio Panach et al., editors, *Proceedings of the ER Forum and Poster & Demos Session 2019, 38th International Conference on Conceptual Modeling (ER)*, volume 2469 of *CEUR WS Proceedings*, pages 113–117, 2019. URL `http://ceur-ws.org/Vol-2469/ERDemo02.pdf`

6. Jan Ladleif and Mathias Weske. A legal interpretation of choreography models. In Chiara Di Francescomarino et al., editors, *Business Process Management Workshops. BPM 2019*, volume 362 of *Lecture Notes in Business Information Processing*, pages 651–663. Springer, Cham, 2019. doi: 10.1007/978-3-030-37453-2_52

7. Jan Ladleif, Mathias Weske, and Ingo Weber. Modeling and enforcing blockchain-based choreographies. In Thomas Hildebrandt et al., editors, *Business Process Management. BPM 2019*, volume 11675 of *Lecture Notes in Computer Science*, pages 69–85. Springer, Cham, 2019. doi: 10.1007/978-3-030-26619-6_7

A further manuscript based on a chapter of this thesis has been prepared:

8. Jan Ladleif and Mathias Weske. Which event happened first? Deferred choice on blockchain using oracles. *CoRR*, abs/2104.10520, 2021. URL `https://arxiv.org/abs/2104.10520`

Acknowledgments

Pursuing a doctoral degree is said to be a journey like no other, and having ventured through it myself now I am inclined to agree. Before and while conducting the work that ultimately resulted in this thesis, I was lucky enough to have a wide network of support. Many thanks are due:

To my supervisor, Mathias Weske, for giving me plenty of room and time to evolve my ideas—and advice on how to narrow them down again, often after fruitful and intense discussions. I enjoyed a degree of freedom and opportunity which should not be taken for granted, and for which I am very grateful.

To Jan Mendling and Cesare Pautasso, for agreeing to review this thesis. Their excellent work on blockchain technology in general and in the Business Process Management (BPM) domain in particular served as a steady inspiration to me.

To Ingo Weber, for introducing me to the topic of blockchain-based process execution. His invitation to join his group in Australia for my Master's thesis sparked a string of ideas and collaboration which principally shaped this dissertation.

To Adriatik Nikaj, for motivating me to actually commence on this journey when I was struggling with the decision. His advice has certainly had an impact and helped me when it felt like there was no end of open questions in sight.

To Anton von Weltzien, for becoming more involved with the development of chor-js than I ever anticipated and contributing significantly to its success. I fondly remember the many evenings spent discussing just how broken the choreography diagram standard is.

To my former and current colleagues, for becoming friends inside and outside of work, divulging in lighthearted gossip, and of course participating in the occasional discussion actually relating to research. In particular, Stephan Haarmann, Simon Remy, Maximilian Völker, and Sven Ihde for proofreading manuscripts of this thesis.

To my family, for making me who I am today and continuing to provide me the best foundation imaginable. To my friends, for distracting me from the constant struggle of writing papers. And to Nicole, for always forcing me to focus again—and offering me room and board.

Contents

List of Figures

List of Tables

Acronyms

ABI	Application Binary Interface
BPMN	Business Process Model and Notation
BPMS	Business Process Management System
BPM	Business Process Management
DApp	Decentralized Application
DLT	Distributed Ledger Technology
DSL	Domain-Specific Language
EVM	Ethereum Virtual Machine
ISO	International Organization for Standardization
LO	Legal Ontology
MDE	Model-Driven Engineering
OMG	Object Management Group
RIM	Railway Infrastructure Manager
SCMS	Smart Contract Management System
UML	Unified Modeling Language
UTC	Coordinated Universal Time
WfMC	Workflow Management Coalition
zkSNARK	Zero-Knowledge Succinct Non-Interactive Argument of Knowledge

Chapter 1

Introduction

Contracts have been an integral part of human life for centuries. Widespread trade and the emergence of services required a common ground as to which actions are appropriate, and which must be performed to the benefit of all involved parties [25]. As society evolved, so did the understanding of such permissions and obligations, of liberty, of law and jurisdiction [48]. While the legal framework surrounding them continued to grow more complex, the written contract as an artifact seemed to withstand change over a long period of time. Whether it is carved in a stone, stamped on a clay tablet, or written on paper; the mainly notarial purpose of contract documents did not significantly change.

In the era of computers, however, this status is finally being challenged. By storing them digitally and interacting with them via the internet, electronic contracts allow humans to get rid of physical contract documents and even automate many aspects of their performance [27]. Consider, for instance, a *contract of carriage*—or train ticket, for short. Germany's national railway company sold around 45 million digital train tickets through their smartphone application in 2019 [29]. Not only did this, according to their calculation, save around 222 tons of paper; rather, it seemingly constitutes a paradigm shift. The contract as well as its state are stored in a digital form, managed by the railway company. To top it off, all contractual interactions—payments and discount applications, cancelations, even the check-in previously done by a conductor within the train—can be performed by electronic means.

And yet, the interactive façade still hides a lengthy contract document written in complex legal language paired with general terms and conditions, devoid of any instructions actionable by an information system [30]. The operational logic is proprietary, largely invisible to the passenger, and centralized within the railway company. While this imbalance of power between the parties may be widely tolerated for inconsequential contracts like a train ticket, many types of agreements do not lend themselves well to this architecture; especially when large sums of money and mutually

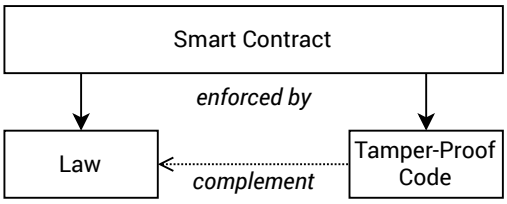


Figure 1.1: Enforcement of a smart contract

distrustful parties are involved. This only becomes obvious when disputes arise and legal mechanisms kick in—for example, when a train is canceled and the passenger sues for a refund.

The vision of *smart contracts* is to close the gap between the specification and the performance of a contract, and to alleviate any imbalance of power between its parties [104]. The core idea is that the contracts themselves are not destined to be passive artifacts. Instead, they could be drafted so that information systems understand their intent and grasp the legal relations between the parties [24]. Instead of the law being the only enforcement contingency, tamper-proof code could enable the autonomous enforcement of certain terms of the legal agreement (see Fig. 1.1). As an effect, the probability a dispute even occurs, let alone grows into a costly lawsuit, would be reduced significantly.

1.1 Research Questions

Smart contracts are still very much a vision, despite the uptake in electronic contracting. A tangible deficiency in practice is certainly the lack of a technological framework to enforce smart contracts in a way that can not be manipulated or rigged by any party or outside entity [24]. Only recently did the introduction of blockchain technology [85] provide new impulses at resolving this issue:

Blockchain technology enables a network of computer nodes to store and share arbitrary information—like account balances and transfers of cryptocurrency—, ensuring its integrity, immutability, and transparency to all network participants [115]. This is achieved using the underlying blockchain data structure, which employs cryptographic techniques combined with an assortment of network protocols to reach consensus on the state of all data. Second-generation blockchain networks like Ethereum [114] extend on these capabilities by introducing Decentralized Applications (DApps): Applications whose code and state are persisted entirely as data in the blockchain. DApps offer fully autonomous and tamper-proof execution of code such that no single network participant yields any elevated rights or may repudiate any of their actions.

Ostensibly, blockchain technology is a perfect fit for smart contracts—so much so that DApps are often also called “smart contracts” themselves, though they do not by default suggest any legal meaning. The research on

how exactly blockchain networks can power smart contracts in the sense introduced above is still in its infancy, however—not least due to the many idiosyncrasies that users face when developing and using DApps [78]: The restrictive runtime environment, co-existing blockchain networks with vastly different properties, as well as high stakes and low tolerance for programming errors. Overall, the development landscape poses many challenges.

The goal of this thesis is to solve some of these challenges, and pave the way towards a structured smart contract specification, management, and enforcement approach that regards blockchain technology and DApps as a first-class citizen and enabler. We will advance the state of the art in this area by tackling the following research questions:

(RQ1) How can a management system for smart contracts built on blockchain technology be designed?

(RQ2) Can DApps correctly and autonomously enforce smart contracts?

Both of these research questions rely on an underlying assumption, which is that there must be a way to fully specify smart contracts formally in the first place; expressive enough to capture all the legal and behavioral facets of smart contracts, while remaining accessible to the involved parties—specifically their human actors.

1.2 Approach

The ultimate enabler of smart contracts seems to be a common modeling language and notation which is actionable for humans of different backgrounds as well as for information systems alike [24]. Such models would be a necessary step for smart contracts to find widespread use and acceptance, since they fully specify smart contracts and at the same time do "not alienate the lawyer" [59]. We adopt this view and follow a Model-Driven Engineering (MDE) approach [99] throughout this thesis:

To this end, we devise a smart contract metamodel capturing the basic legal and structural aspects of smart contracts as derived from literature (see Fig. 1.2). We will formally define the operational semantics of the resulting smart contract models, including a notion of autonomous behavior which is peculiar to smart contracts. While the metamodel may be used to assess concrete modeling languages as to their suitability for smart contract modeling—which we will show on an example from the Business Process Management (BPM) domain—or even create new languages, its main purpose is to guide our approach:

We use the smart contract metamodel to design a management and execution platform for smart contracts, and argue about the distribution of their state and features required for their proper enforcement (RQ1).

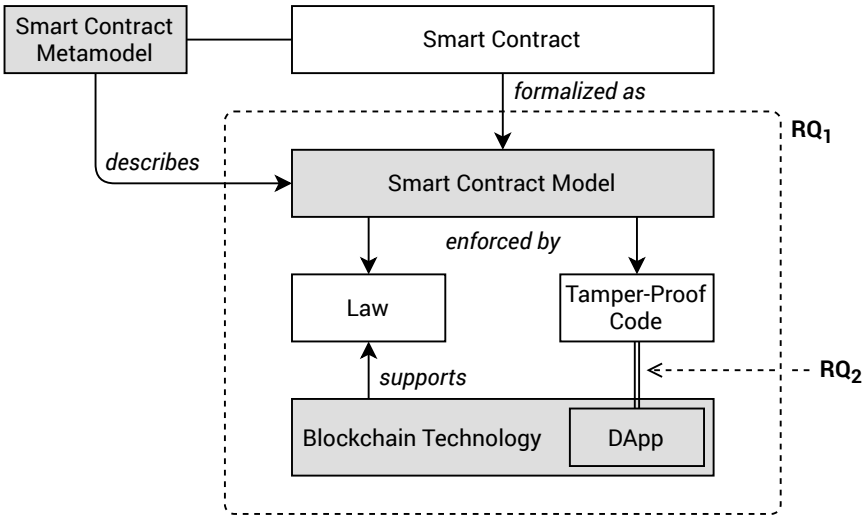


Figure 1.2: Overall approach of this thesis (shaded background) integrated with the structure of smart contracts

Furthermore, we transfer the operational semantics of smart contracts to the peculiar properties of DApps like their transaction-driven runtime and their protocol-induced isolation. The goal is to explore whether a faithful implementation of the semantics is possible and feasible (RQ2).

As a theme throughout this thesis, we will draw inspiration from the BPM domain [112, 31]. Ultimately being concerned with behavioral models as well, and running into similar issues when modeling and executing processes, we aim at leveraging existing work and draw from decades of expertise which has yet to be established for smart contracts. Indeed, there is a growing interest in blockchain technology from the area in its own right, which complements our efforts directly [83].

1.3 Thesis Structure

The thesis is structured as follows:

Chapter 2 **(Preliminaries)**
: This chapter conveys the fundamental knowledge required to appreciate the main contents of this thesis. First, the notion of legal contracts, their formalization, and how they eventually lead to smart contracts are outlined. Second, blockchain technology and DApps are introduced, with a focus on the latter's capabilities and restrictions. Lastly, the domain of BPM is briefly introduced, with a focus on the areas that overlap with how smart contracts may be managed and enforced.

Chapter 3 **(Smart Contract Modeling)**
: In this chapter, a smart contract metamodel is compiled which encapsulates the major smart contract components and their relation-

ships. The operational semantics are introduced and explained using a concrete example model which is subsequently used throughout the thesis. Lastly, the metamodel is used to assess the suitability of Business Process Model and Notation (BPMN) choreography diagrams for modeling smart contracts.

Chapter 4 **(Smart Contract Management Systems)**
Based on the requirements the smart contract metamodel and semantics pose, the Smart Contract Management System (SCMS) approach is proposed in this chapter. The SCMS draws inspiration from the analogue Business Process Management Systems (BPMSs) from the BPM domain, in particular when it comes to transferring existing enforcement approaches to blockchain networks and DApps. The general design principles as well as functional requirements of SCMSs are then discussed in detail.

Chapter 5 **(Proof-of-Concept Implementation)**
In this chapter, parts of the SCMS architecture are implemented in a prototype called Mantichor. This includes a web-based frontend application allowing users to model smart contracts as BPMN choreography diagrams, which are then deployed and enforced by DApps using adapters to two different blockchain networks. In this chapter, the choreography diagram modeling framework chor-js, which was implemented as part of this thesis and has since gained some recognition in its own right, is also briefly presented.

Chapter 6 **(Autonomous Actions on Blockchain)**
Being able to autonomously enforce certain parts of the contractual agreement is one of the major novelties of smart contracts. DApps, however, possess properties which make it difficult to correctly implement smart contract semantics without working around restrictions or sacrificing some of their integrity guarantees. In this chapter, several such approaches are introduced and evaluated as to their feasibility in practice.

Chapter 7 **(Time on Blockchain)**
Time is an important aspect for contracts, since terms and actions are often tied to specific temporal constraints. However, the peculiar properties of blockchain technology often prohibit the use of traditional methods of telling the time in computer systems. In this chapter, the aspects of time in blockchain networks and methods for enforcing temporal constraints autonomously in DApps are explored.

Chapter 8 **(Related Work)**
While smart contract enforcement using blockchain technology is still in its infancy, there is already an expanding body of related

work. The most important approaches from literature will be presented, including also the BPM domain, and compared to the work done in this thesis.

Chapter 9 **(Conclusion)**
In the concluding chapter, the results of this thesis are discussed in detail, with a critical focus on whether the research questions could be answered. Furthermore, future challenges and remaining gaps in research which would warrant a closer look in the future are identified.

Chapter 2

Preliminaries

This thesis is built on a substantial body of existing literature as well as on research that is still ongoing. In this chapter, the main ideas and definitions which shaped our work and are critical for its general understanding are introduced.

We present a notion of smart contracts in Sect. 2.1, focusing on its origins in law. Decentralized Applications (DApps) are introduced in Sect. 2.2 alongside the foundations of blockchain technology. Lastly, we present certain areas of Business Process Management (BPM), in particular the modeling of choreographies, in Sect. 2.3.

In this chapter and the remainder of this thesis, we will use standard mathematical notation for our formal definitions [53], with some exceptions and additions: $[a, b] := \{x \in \mathbb{N} \mid a \leq x \leq b\}$ denotes the inclusive interval from $a \in \mathbb{N}$ to $b \in \mathbb{N}$ in the natural numbers. Given a set S, then $\mathcal{P}(S)$ denotes its power set, and S^* denotes the set of all tuples of arbitrary length over S. Given a tuple $T = (s_1, ..., s_n) \in S^*$, we say $s \in T$ if and only if s is an element of the tuple, that is, $\exists i \in [1, n] : s_i = s$. An empty tuple is equal to the empty set $\emptyset$.

2.1 Smart Contracts in Law

Legal contracts are a fixture in law, and endeavors towards their formalization eventually led to the common notion of smart contracts.

2.1.1 Legal Contracts

Legal contracts are a standard part of every lawyer's repertoire, and even though countless contracts are formed each day, their exact definition and meaning is surely a matter of far-reaching philosophical considerations [25]. In this thesis we follow current practices in law as documented by the widely used reference work Black's Law Dictionary, and adopt a

two-layer view: the contract document on the one, and the binding legal agreements it represents on the other hand [37].

Definition 1 *(Legal Contract).* A legal contract is "an agreement between two or more parties creating obligations [and other legal relations] that are enforceable or otherwise recognizable at law." [37] ◇

To avoid ambiguity in the remainder of this thesis, we will consistently refer to those types of contracts with the qualifier "legal" in distinction to "smart" contracts.

A legal contract is intangible in nature and does not necessarily possess any physical representation. It can be formed orally, for example. The parties to a contract—both individuals as well as entities and organizations—usually have an interest in fixing the terms of the contract before it becomes legally binding, though. This is done using contract documents:

Definition 2 *(Contract Document).* The contract document puts in writing, either digitally or physically, the terms of a legal contract between two or more parties. ◇

The contract document may also be used as a template, being customized and personalized for specific scenarios and participants. A legal contract is entered when all parties sign or otherwise acknowledge the contract document in a way that demonstrates their intention to do so.

Legal contracts and contract documents must be viewed in the context of their local jurisdiction and local customs, that is, it must be "valid under the law of the residence of the party wishing to enforce the [legal] contract" [37]. The same contract document may have vastly different interpretations depending on which country or region it is drafted and entered in. Laws govern how contracts should look, which clauses they may contain, and which implicit terms, for example regarding customer protection, are always required. These overarching rules are also called *meta-rules* [59].

As already insinuated, there are several steps to take in order to enter a legal contract. It does not end there, though, as all parties still need to perform their obligations, disputes need to be resolved, and eventually the legal contract may terminate. The different stages in a legal contract's existence are subsumed in the contract lifecycle. In this thesis, we adopt the contract lifecycle as presented by Governatori et al. (see Fig. 2.1), which has the following phases [39]:

Negotiation
: The parties gather and take the decision to enter into an agreement in pursuit of a common goal. The terms of the agreement are discussed to make sure all parties eventually agree to enter into the agreement.

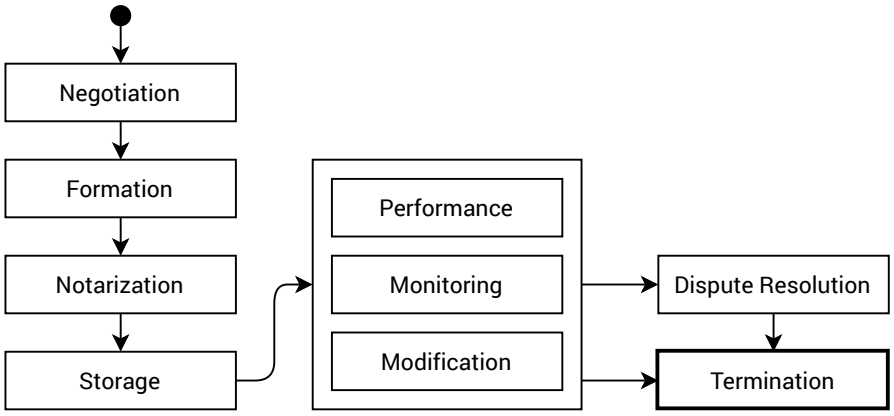


Figure 2.1: Lifecycle of a legal contract as adapted from Governatori et al. [39]

Formation
: The negotiated terms are fixed in legal writing as a contract document that may cross-reference laws or general terms and conditions.

Notarization
: The finished contract document is verified and signed. Possibly, witnesses are present to attest that all parties intentionally entered the ensuing legal agreement.

Storage
: The contract is stored and remains accessible for reference.

Performance
: The parties perform the actions necessary to fulfill their obligations as part of the legal contract.

Monitoring
: To make sure that all parties correctly and honestly perform the contract, mutual monitoring is used. In case of breaches of contract countermeasures may be taken, that is, modification, dispute resolution, or termination.

Modification
: The legal contract may be appended and modified freely as long as all parties agree, for example in reaction to changed circumstances.

Dispute Resolution
: In case of conflicts which can not be easily resolved the parties may seek an arbitration or litigation process or find a mutual way to settle the legal contract.

Termination
: Lastly, a legal contract is terminated when the parties have fulfilled

all of their obligations, or agreed to end their relationship prematurely.

Of course, these phases are not necessarily in order or even exclusive to each other, as shown in the flowchart in Fig. 2.1. However, they give an intuition as to which phases a legal contract goes through from negotiation to termination. We will use the contract lifecycle throughout the thesis for both legal as well as smart contracts—although the adoption of smart contract may eventually yield new or altered phases, current research suggests that the contract lifecycle remains largely applicable as of today [39].

2.1.2 Formalization of Legal Contracts

A contract document is, for all intents and purposes, a formal document owing to its complicated rule-based structure and language. *Legal English*, or legalese [37], is often regarded to be its own language entirely, having distanced itself from colloquial and spoken English over centuries. In the context of smart contracts, however, formalization rather refers to achieving a degree of mathematical preciseness readable and analyzable for machines [105].

Legal Relations

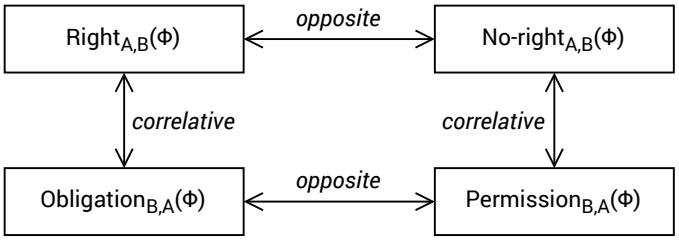
When parties enter into a legal contract, they are subject to the legal relations given by the terms of the contract document. There are various types and categorizations of legal relations in literature [27]. In this thesis, we adopt the seminal categorization of legal relations (see Fig. 2.2) according to Hohfeld, who discerns norms of conduct and norms of power [48]:

Definition 3 *(Norms of Conduct).* Let A and B be two parties to a contract, and Φ be an action describing some concrete behavior or interaction of A or B. Then there are four *norms of conduct* relating to Φ:

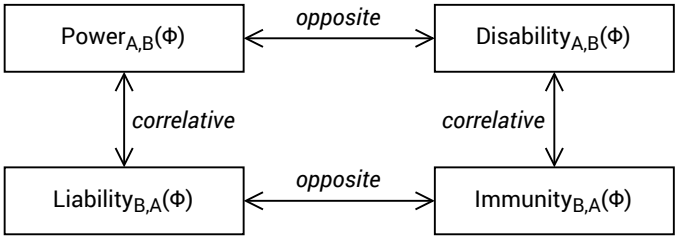
- $\mathsf{Right}_{A,B}(\Phi)$, the right of A that B performs Φ,
- $\mathsf{Obligation}_{A,B}(\Phi)$, the obligation of A to perform Φ in face of B,
- $\mathsf{No\text{-}right}_{A,B}(\Phi)$, the no-right of A to Φ performed by B, and
- $\mathsf{Permission}_{A,B}(\Phi)$, the permission of A to perform Φ in face of B.

Some authors additionally recognize the notion of liberties, $\mathsf{Liberty}_{A,B}(\Phi) \equiv \mathsf{Permission}_{A,B}(\Phi) \wedge \mathsf{Permission}_{A,B}(\neg\Phi)$, which emphasize that a party has the express choice to accept or decline performing an action [40, 6]. ⋄

Norms of conduct are fundamentally related to each other as shown in Fig. 2.2a. A correlative legal relation is one that must exists in conjunction with another. For example, the right $\mathsf{Right}_{A,B}(\Phi)$ of A to an action Φ against



(a) Norms of conduct



(b) Norms of power

Figure 2.2: Legal relations according to Hohfeld [48]

B directly implies there is an obligation $\text{Obligation}_{B,A}(\Phi)$ of B to perform or allow the performance of said action Φ against A. The same is true for no-rights and permissions, respectively. Further, legal relations can be opposites. A right is the opposite of a no-right, and an obligation is the opposite of a permission.

In addition to norms of conduct, there are norms of power. They characterize situations in which a participant may extinguish and create legal relations:

Definition 4 *(Norms of Power).* Let A and B be two parties to a contract, and Φ be an action describing some concrete behavior or interaction of A or B. Then *norms of power* are legal relations allowing a party to create, modify, and extinguish other legal relations. There are four norms of power:

- $\text{Power}_{A,B}(\Phi)$, the power of A to impose Φ on B,
- $\text{Liability}_{A,B}(\Phi)$, the liability of A to be imposed to Φ by B,
- $\text{Disability}_{A,B}(\Phi)$, the disability of A to impose Φ on B, and
- $\text{Immunity}_{A,B}(\Phi)$, the immunity of A to be imposed to Φ by B. ◇

Norms of power are similarly connected to each other as norms of conduct, a power being correlative to a liability and a disability being correlative to an immunity (see Fig. 2.2b). That is, for example, if A has a power $\text{Power}_{A,B}(\Phi)$ over B regarding an action Φ, then B is liable as to that action. For example, a common term in contracts is that a party is liable for all damages should they occur.

More elaborate systems to capture legal relations have been proposed. For example, Crawford and Ostrom extend legal relations to institutions, and specify them in the so-called ADICO format [28]: *Attributes* specify for whom the legal relation holds, which is represented by a *deontic* modifier expressing the proposed conduct towards a concrete *aim*—the "I" in ADICO—, under certain *conditions* and with *or else* sanctions. More recently, Pace and Schneider developed a language for composing and arguing about deontic notions [89].

Legal Ontologies

Ontologies are a means of structuring knowledge in certain areas of interest, and are an important gateway to mediate semantics and meaning between human beings and machines [19]. In law, Legal Ontologies (LOs) capture legal concepts:

The Legal Knowledge Interchange Format (LKIF) approach introduces a multi-level core ontology used for interchanging generic legal information between different legal knowledge systems [47]. Kabilan and Johannesson use a similar multi-tier ontology framework for modeling contract knowledge for the semantic web [56]. Legal contracts are modeled as Unified Modeling Language (UML) class diagrams on three layers relating to the high-level contract terms down to a template view with some support for performance and enforcement concepts.

More specialized approaches such as the Unified Foundational Ontology for Legal Relations (UFO-L) embrace standard deontic notions as posed by Hohfeld [48] and Alexy [6] and focus on small subsets of knowledge, in this case legal relations and how they relate parties [40]. There is no concept of time or the passing of the same; instead, each UFO-L instance represents a static snapshot of the situation at a concrete point in time.

LOs lift the legal meaning and the legal implications of a legal contract as posed by the contract document from their prosaic form to a format accessible by machines. This makes them an essential help in conceptualizing and modeling smart contracts, especially since they abstract from complex and non-obvious concepts like open-textured terms in legal prose such as "without delay", which are difficult to capture precisely and not are immediately enforceable by machines [39].

2.1.3 Smart Contracts

As the name suggests, smart contracts not just correspond to a legal agreement like legal contracts, but provide some additional value or features. The initial idea for smart contracts dates back decades when electronic contracting was becoming more and more commonplace [73, 104]. Most of these early notions of smart contracts rely on two observations:

First, that the validity and performance of legal contracts can be enhanced via cryptographic methods. For example, a contract document may

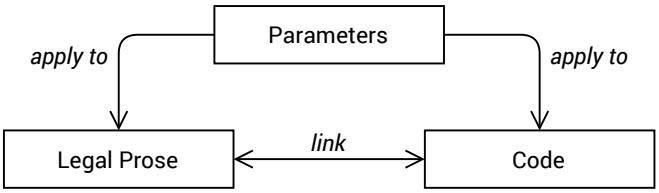


Figure 2.3: Components of the Ricardian triple

be electronically signed and encrypted to exchange it digitally, or the performance of certain actions may be certified digitally. Grigg introduced the concept of the *Ricardian contract* in an early electronic payment system based on this observation [41]. Basically, a Ricardian contract is a text file that is cryptographically signed by a legal issuer and expresses some value to its holders.

Second, that legal contracts contain operational aspects which can be automated using software systems. For instance, recurring payments as part of a subscription are often susceptible to this. The Ricardian contract was further refined to this end to form *Ricardian triples* (see Fig. 2.3), which propose three major components [24, 23]:

Code
: Code is written in an executable programming language, possibly a Domain-Specific Language (DSL), and allows software systems to automate certain parts of the underlying legal contract. This component captures the *operational* aspects of legal contracts.

Legal Prose
: Some clauses of legal contracts, such as open-textured terms or basic norms of power mentioned above, do not necessarily contain immediately actionable descriptions of behavior for a machine. Such *non-operational* aspects are intangible, ambiguous, or interpretive in nature.

Parameters
: Lastly, parameters represent the results of negotiations on the level of a concrete, binding agreement. The code and legal prose components may be seen as templates, which get finalized for a concrete purpose by fixing the parameter values, e.g., the price of a train ticket or a custom deadline.

Figure 2.3 shows the interplay of the three components. Legal prose and code are interlinked in some way, e.g., by references or hyperlinks, and are both influenced by parameters. This model, lastly, mainly influenced the general definition of smart contracts by Clack et al., which we will use throughout this thesis when referring to smart contracts:

Definition 5 *(Smart Contract).* "A smart contract is an automatable and enforceable agreement. Automatable by computer, although some parts

may require human input and control. Enforceable either by legal enforcement of rights and obligations or via tamper-proof execution of computer code." [24] ◇

The existence of smart contracts has manifold implications for the legal domain, since a rising degree of automation and enforcement via code would in the long run diminish the importance of legal enforcement. From a conceptual point of view, legal experts also differ on whether smart contracts actually fit the definition of a legal contract, alluding primarily to whether they properly match the contract lifecycle and fulfill the legally required steps to enter into a binding agreement [111].

Some of this debate may stem from the conflation of DApps, which are also sometimes called smart contracts [114] (see Sect. 2.2), with the legal concept of a smart contract as described above. Whereas the former is ultimately a means to implement the tamper-proof enforcement portion of the smart contract, the latter is strictly aligned with legal contracts in itself. It is out of scope of this thesis to assess any philosophical conflicts regarding the exact placement of smart contracts within the legal domain.

2.2 Blockchain Technology

The term *blockchain technology* encapsulates the frameworks and protocols surrounding the implementation and use of blockchains as introduced by Bitcoin [85], and is considered to be a concrete example for the more general Distributed Ledger Technology (DLT). Blockchain technology has since grown from merely handling cryptocurrency to supporting the trading of generic assets, and managing DApps, applications whose entire code and state is persisted as data within a blockchain [18].

In this section, we provide a high-level overview and formalization of a common architecture for blockchains and blockchain networks, as well as the lifecycle of transactions. The formalization is then used to introduce the concept of DApps and their peculiar properties, along the widely-used oracle pattern to deal with the isolation of the blockchain network.

2.2.1 Blockchain Data Structure

The fundamental data structure at the core of blockchain technology is the blockchain, which provides an append-only data store built around transactions. Blockchains have an innate temporal aspect, which we will capture using a discrete time model of timestamps:

Definition 6 *(Timestamp).* A *timestamp* $t \in \mathbb{N}$ specifies a distinct point in time. An increased timestamp $t + 1 \in \mathbb{N}$ marks the passing of a uniform unit of time. ◇

We specifically chose the natural numbers $\mathbb{N}$ as our time domain in this thesis for its simple arithmetic properties, and the fact that most computer

systems rely on a notion of time which directly maps to $\mathbb{N}$, e.g., the number of seconds since the Unix epoch for Unix time. Then a blockchain is defined as follows:

Definition 7 *(Blockchain).* A blockchain $\mathcal{B} = (B_0, ..., B_n)$ is a cryptographically linked list of n numbered blocks. A block $B_i = (t_i, T_i)$ with block number i contains

- the *block timestamp* $t_i \in \mathbb{N}$, which strictly increases from the previous block, $t_{i-1} < t_i \, \forall \, i \in [1, n]$, and
- an ordered *list of transactions* $T_i \in \mathbb{T}^*$ where $\mathbb{T}$ is the set of all transactions, that is, instructions to the blockchain.

The first block B_0 is called the *genesis block* and does not contain any transactions, i.e., $T_0 = \emptyset$. ◇

In practical implementations, blocks are limited in how many transactions they can contain for storage size reasons. The delay between subsequent blocks, sometimes called block time, thus mainly determines how many transactions and how much data the blockchain can hold in total:

Definition 8 *(Block Delay).* Let B_i with $i > 0$ be a block of a blockchain $\mathcal{B} = (B_0, ..., B_n)$ other than the genesis block. Then the block delay $d_i := t_i - t_{i-1}$ is the amount of time passed between block B_i and its predecessor. ◇

The resulting blockchain structure is shown on a timeline in Fig 2.4. Using a combination of cryptographic methods, the blockchain data structure guarantees integrity of its contents. Each block on its own can be validated very quickly depending on the concrete implementation, e.g., by computing the number of trailing zeroes of its hexadecimal hash value against a network-mandated threshold in Ethereum [114]. Inversely, this makes it exceedingly expensive from a computational perspective to create valid new blocks, as they can only be found by repeatedly changing a specific section of the block until a fitting hash is produced. This reduces the probability of an adverse agent being able to manipulate blocks.

This integrity mechanism in conjunction with the cryptographical link connecting a block and its predecessor, i.e., storing the previous block's

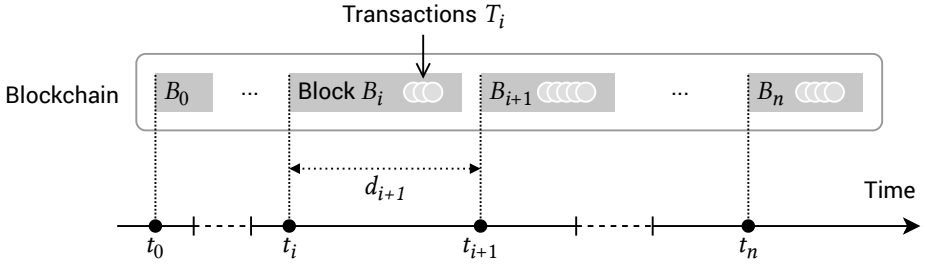


Figure 2.4: Structure of a blockchain

hash, effectively make blockchains immutable. It is easily detectable that a blockchain has been tampered with by validating the integrity of all blocks, as any change in a block breaks the chain of hashes. Thus, the only way of "adding" data to or "modifying" data in a blockchain is by appending a new block to its end, essentially creating an extended blockchain. This procedure is generally called mining:

Definition 9 *(Mining).* Let $\mathcal{B} = (B_0, ..., B_n)$ be a blockchain and $B_{n+1} = (t_{n+1}, T_{n+1})$ be a new block. Then the new block can be appended to $\mathcal{B}$ forming a new, extended blockchain $\mathcal{B}' = (B_0, ..., B_n, B_{n+1})$, iff

- B_{n+1} fulfills all integrity requirements, i.e., it has a valid hash and includes a hash of the previous block B_n,
- $t_n < t_{n+1}$, that is, the new block was created after B_n, and
- all transactions $tx \in T_{n+1}$ are properly signed and valid with respects to $\mathcal{B}$, i.e., there is no previous transaction which in any way conflicts with tx. ◇

Transactions may have different purposes, from simply storing some attached payload data to performing calculations on already existing data. For example, initially storing the information attached to a train ticket in a blockchain may be one transaction, and setting its validity state to "used" may be another. The overall state of the data stored in the blockchain is thus given by the subsequent application of all contained transactions in order.

In the scope of this thesis, we abstract from how data is attached to transactions and how the stored data can be addressed in detail. Blockchain implementations like Ethereum, for instance, will employ data structures like Merkle trees which allow for efficient storage and retrieval of data without having to replay all transactions constantly [114].

2.2.2 Blockchain Networks

Blockchains are stored and distributed within networks. Such a blockchain network is formed by a set of nodes run by individuals, organizations, or other entities who have a common interest in maintaining some data using a blockchain, e.g., a ledger of account balances of some cryptocurrency like Bitcoin.

Definition 10 *(Blockchain Network).* A *blockchain network* $\mathcal{N} = (N, A)$ consists of a set of nodes N, which are linked to form a connected network, and accounts A. Each node owns a replica of the *currently accepted blockchain* $\mathcal{B}_{\mathcal{N}}$. Further, a global *transaction pool* $T_{\mathcal{N}} \subseteq \mathbb{T}$ holds transactions which are signed by an account and pending to be included in a subsequent extended blockchain. ◇

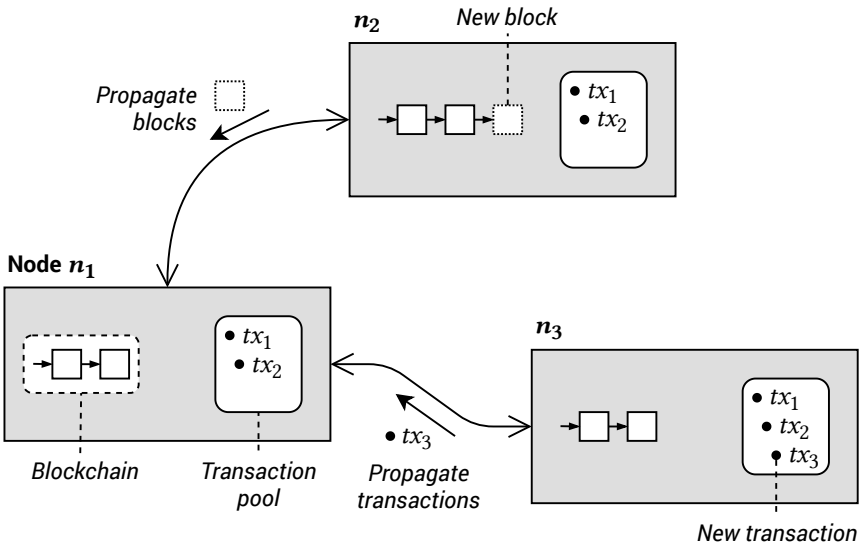


Figure 2.5: Excerpt from a blockchain network showing three nodes and their connections, as well as block and transaction propagation

Blockchain networks are inherently distributed systems and there is no central authoritative entity (see Fig. 2.5). The above definition abstracts from some of the resulting complications: For one, each node actually maintains its own transaction pool and new transactions are propagated within the network, only virtually achieving a global transaction pool $T_{\mathcal{N}}$.

Arguably the major contribution of blockchain technology in general is consensus, that is, figuring out exactly which blockchain $\mathcal{B}_{\mathcal{N}}$ is currently accepted across the network. Such consensus is driven by the integrity and immutability properties of the blockchain data structure. Starting from an initial empty blockchain containing only the genesis block, nodes take on the role of *miners*:

They collect transactions from the transaction pool and assemble them into a new block, yielding an extended blockchain $\mathcal{B}'$. The new block is propagated within the blockchain network, validated by other nodes, and $\mathcal{B}'$ is eventually accepted by a majority of the network as the current blockchain $\mathcal{B}_{\mathcal{N}}$. All of this is backed by an incentive mechanism, which rewards miners of new blocks with cryptocurrency partly generated and partly paid for by the senders of transactions.

It could happen that multiple conflicting blockchains exist simultaneously if miners create and propagate new blocks at the same time. These conflicting blockchains are called *forks*, and one needs to be chosen above all others eventually. One approach is to accept longer blockchains over shorter blockchains based on the number of blocks, which makes it unlikely that forks exist for an extended period of time. Still, a practical consequence is that recently included transactions may be "removed" by switching to a fork, which is why it is advisable to wait for a certain period of time, the *confirmation time* (see Sect. 2.2.3), until acting upon a transaction [117].

Of course, the above formalization is an abstraction. Many consensus algorithms like Proof-of-Work (PoW), Proof-of-Stake (PoS), and Practical Byzantine Fault Tolerance (PBFT) exist, which dictate how the network approaches mining, block validation, forks, and other things in detail [16]. More traditional blockchains like Bitcoin [85] and Ethereum [114] follow a very transparent approach based on equal rights, that is, all nodes N have full access to all the blockchain's contents and pending transactions, and can participate in the mining procedure. In practice, the strictly public nature of blockchain networks is often not desirable for organizational or confidentiality reasons, which is why permissioned blockchain networks were devised [90]:

Definition 11 *(Permissioned Blockchain Network).* A blockchain network $\mathcal{N} = (N, A)$ is *permissioned*, if some nodes in N have elevated rights over the others. These rights may, for example, pertain to submitting transactions, participating in the mining and consensus procedures, or accessing data stored in the blockchain. ◇

Permissioned blockchain networks, like those built on Corda [45] or Hyperledger [9], sacrifice some of the fundamental properties of their public counterparts, invoking disparity and the need for trust between nodes. *Consortium blockchain networks* are an example for permissioned blockchain networks, e.g., formed by a consortium of railway companies to manage cross-border ticketing. In most settings, only nodes part of the consortium would be allowed to append new transactions to the blockchain. Some consortium blockchain networks may even be hidden entirely from non-privileged members.

2.2.3 Transaction Lifecycle

As already hinted above, a transaction $tx \in \mathbb{T}$ passes through several phases before it is safely included in the currently accepted blockchain $\mathcal{B}_{\mathcal{N}}$ of a blockchain network $\mathcal{N} = (N, A)$. These phases are subsumed in the *transaction lifecycle*. While there are several views on which phases the transaction lifecycle should include depending on which blockchain network it captures, we adopt a very simple and minimal version loosely based on Xu et al. [117] containing only four phases (see Fig. 2.6):

Created
: The sender locally created the transaction tx. It encapsulates the payload and all other necessary information according to the intent of the sender in the format required by the target blockchain network. Further, tx is signed using a blockchain account from A belonging to the sender.

Submitted
: The sender submitted tx to the blockchain network via a block-

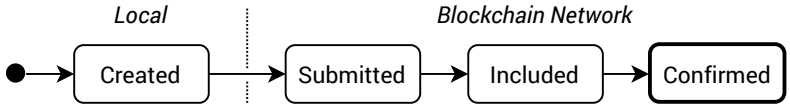


Figure 2.6: Lifecycle of a transaction

chain node. The transaction tx has subsequently been propagated and placed in the global transaction pool, i.e., $tx \in T_{\mathcal{N}}$.

Included
: The transaction tx was picked up by a miner and has been included in an extended blockchain $\mathcal{B}'$.

Confirmed
: The extended blockchain $\mathcal{B}'$ containing tx is widely accepted as the current blockchain $\mathcal{B}_{\mathcal{N}}$ within the blockchain network, and it is improbable that it will be displaced by the emergence of a longer fork. When this phase is reached depends on the individual risk tolerance of the sender.

The transaction lifecycle gives rise to a number of timestamps characterizing a single transaction:

Definition 12 *(Transaction Timestamps).* The exact time at which a transaction $tx \in \mathbb{T}$ entered the Created, Submitted, Included, or Confirmed lifecycle state is referred to as the creation, submission, inclusion, or confirmation timestamp of tx, respectively. We will refer to the submission timestamp simply as the *transaction timestamp* $t_{tx} \in \mathbb{N}$. ◇

Note that this lifecycle mirrors only the scenario in which a transaction is successfully confirmed eventually. In practice, transactions may be dropped or rejected by the network for several reasons, which has to be considered by a sender [117]. Also, senders have to take into account the variable delays between the lifecycle phases, which depend on the average block time of the blockchain network and the monetary incentives associated with a transaction [118]. In this context, the inclusion delay is an important measure:

Definition 13 *(Inclusion Delay).* Let $tx \in \mathbb{T}$ be a transaction contained in a block B_i of a blockchain, i.e., $tx \in T_i$. Then the *inclusion delay* $d_{tx} := t_i - t_{tx}$ of tx is the delay between its submission timestamp t_{tx} and its inclusion timestamp, which is equal to its including block's timestamp t_i. ◇

2.2.4 Decentralized Applications

Many second-generation blockchain networks introduced the notion of DApps under different names: "smart contracts" in Ethereum [114] and Tezos [38], "chaincode" in Hyperledger [9], and "CorDapp" in Corda [45],

to name a few. For disambiguation reasons, especially in distinction to smart contracts as a legal concept, we will consistently refer to them as DApps in this thesis, and define them as follows:

Definition 14 *(Decentralized Application).* A Decentralized Application (DApp) Ð is a uniquely identified computer program whose code and state is entirely persisted as data in a blockchain $\mathcal{B}$. DApps are called using transactions and executed during the mining procedure. Their interface functions as well as their input and output parameters are stated in the Application Binary Interface (ABI) of the DApp. ◇

DApps are developed using general-purpose programming languages like Java, or domain-specific languages like Solidity tailored to certain blockchains. Interacting with a DApp is inherently asynchronous: A user sends a transaction, and only receives a result once the transaction has been successfully included within the blockchain network.

A DApp Ð needs to be initially deployed to a blockchain $\mathcal{B}$ using a special kind of transaction carrying the code as its payload. During the mining process, this code is added to the blockchain, Ð is initialized, and a unique identifier—called *address*—is created, which must be used to identify Ð in future transactions. The address is also used by other DApps to interact with Ð, allowing the development of complex interacting systems.

DApps must adhere to a number of restrictions, which we express in three fundamental properties:

Immutability Property

As the code of a DApp Ð is stored as data in the blockchain and blockchains are inherently immutable, it can not be changed once deployed. While this can certainly be regarded as a feature, assuring participants that Ð can not be manipulated, it also places high demands on the maturity of individual DApps intended for use in production.

This has to be taken into account in all stages of DApp development. Verification techniques like model-checking may be used to ascertain that no mistakes or unintended functions were introduced [78]. It is also possible to use proxy and adapter patterns involving multiple linked DApps to allow replacement of certain components by changing the pointers between them, which are part of the state and not the code of the DApps [61].

Isolation Property

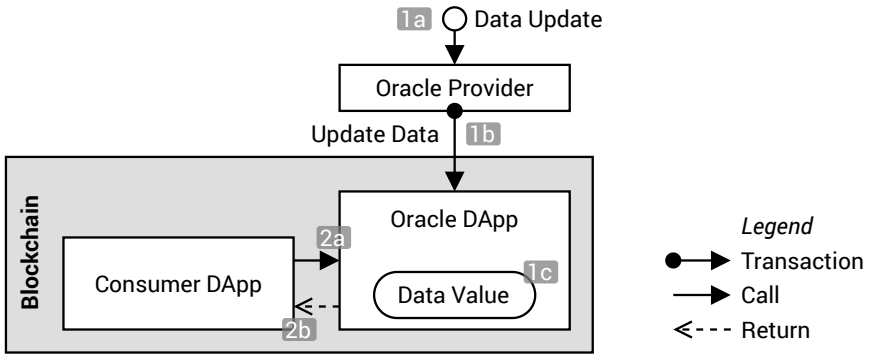
Every block of a blockchain, including transactions and other data, needs to be independently verifiable by all nodes of a blockchain network to achieve integrity [115]. This is only guaranteed to be possible if all the data a DApp uses is contained within the blockchain. This directly results in a strict isolation property: DApps can not directly access data or services outside the blockchain, since those may be tampered with or may cease to exist.

Non-Continuity Property

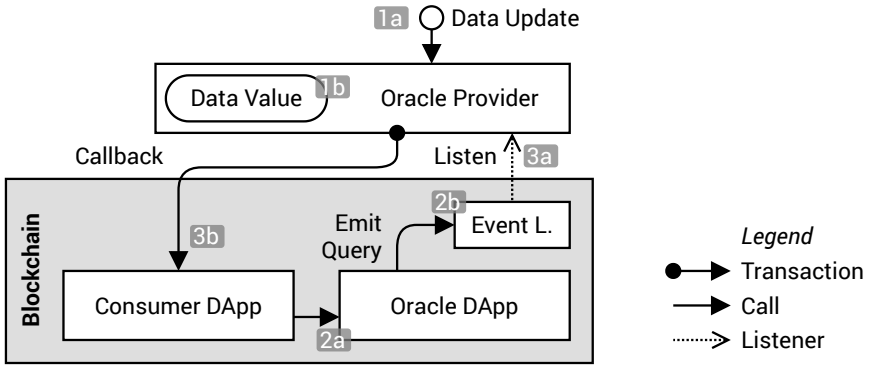
A DApp Ð is only executed during the mining procedure when an atomic transaction tx targeting one of its exposed functions according to the ABI is being included. Thus, Ð essentially lies dormant between subsequent blocks. That is, neither can its state be changed between blocks, nor can it implement continual behavior such as busy waiting or polling. Algorithms requiring these kinds of behavior need to be adapted to fit this transaction-driven execution scheme.

2.2.5 Oracle Patterns

DApps may require access to data that is maintained outside of the blockchain despite the isolation property, for example currency exchange rates or timestamps provided by a time server. Oracles are a means to allow access to such external data from within a DApp [116], and many variants of the pattern have since been devised [84, 80, 113]. The idea is not to break the isolation property, but to circumvent it using the regular mechanisms available to DApps.



(a) Storage oracle



(b) Request-response oracle

Figure 2.7: Architecture and behavior of the storage and request-response oracles

One of the most well-known types of oracles is the *storage oracle* (see Fig. 2.7a). Here, an external oracle provider, which can be an organization or a software system maintained by some entity operating independently of the blockchain network, constantly keeps track of the external data. When the data is updated and its value changes (1a), the oracle provider sends a transaction containing the new value as the payload to a dedicated oracle DApp (1b). The oracle DApp stores this value as part of its state on the blockchain (1c). The exact address of the oracle DApp is publicly known, and it provides interfaces to acquire the most recent value of the data it possesses. As such, consumer DApps can query the oracle DApp (2a), which immediately returns the queried value (2b).

The second oracle type is the *request-response oracle* (see Fig. 2.7b). Again, there is an oracle provider and an oracle DApp. In this case, though, the oracle provider reacts to updates of the data (1a) by storing the value locally and thus off-chain (1b). To acquire the current value consumer DApps call the oracle DApp (2a), which then emits a query using the blockchain network's event mechanism (2b). Such an event, in short, can be regarded as a flag that outside entities may observe. The event further contains the address of the consumer DApp. The oracle provider picks up the event (3a), and sends the value of the data directly to the consumer DApp in a new transaction (3b).

Actual oracles may of course deviate from these patterns. OrFeed[1], for instance, provides current cryptocurrency exchange rates using a storage oracle approach, extended by paid tiers of subscription services. Maker[2] provides similar services, but adds a way of aggregating oracle values in a so-called "medianizer" component. Provable[3] is perhaps the most well-known request-response oracle, including optional verification features and the option to query arbitrary external services by providing a URL. Oracle providers may also organize in networks themselves, to avoid concentration of power in a single oracle provider [4]. A survey of these and other approaches has recently been given by Al-Breiki et al. [5].

2.3 Business Process Management

Businesses offer services or products to their customers, which rely on certain activities being performed both internally and in exchange with external partners. These activities follow explicit or implicit guidelines, generate data artifacts, and are restrained by rules; all things considered forming a business process. BPM encapsulates the large body of methods, research, and techniques towards identifying, modeling, enacting, and evaluating those business processes [112]. In this section, selected concepts used in

[1] `https://www.orfeed.org/`
[2] `https://developer.makerdao.com/feeds/`
[3] `https://provable.xyz/`

BPM are introduced, with a particular focus on business process choreographies.

2.3.1 Business Processes

A business process is characterized by a set of activities, which are performed in a coordinated fashion within one organization [112]. The activities may be performed by human actors or software systems, and collectively work towards reaching a business goal. The goal of BPM is to make processes palpable for all stakeholders, to describe them in an understandable way, analyze as well as optimize them, and ultimately automate them. This is generally done using models:

Definition 15 *(Process Model).* A process model is a blueprint for processes containing specifications of activities and their coordination. Coordination often pertains to a causal ordering of activities, but may also be contingent on process data [94], resource usage and availability [95], as well as complex event and workflow patterns [96]. ◇

In BPM, the introduction of processes in organizations follows a certain structure, the so-called *business process lifecycle*. Figure 2.8 shows this lifecycle as described by Weske [112].

The first phase is the combined *design and analysis* phase. Here, the business process is properly identified and modeled in the design step. The result is a process model, which serves two major purposes: It formally specifies all components and activities of the process, and can be visualized and displayed in various degrees of abstraction to convey the process to its stakeholders. In the analysis step, validation, simulation, and verification techniques are applied to the business process. These may bring to light formal or compliance issues which can then be resolved.

In the *configuration* phase, the automation environment is configured to support the automation of the process. This includes selecting a fitting system infrastructure, implementing parts of the process in computer

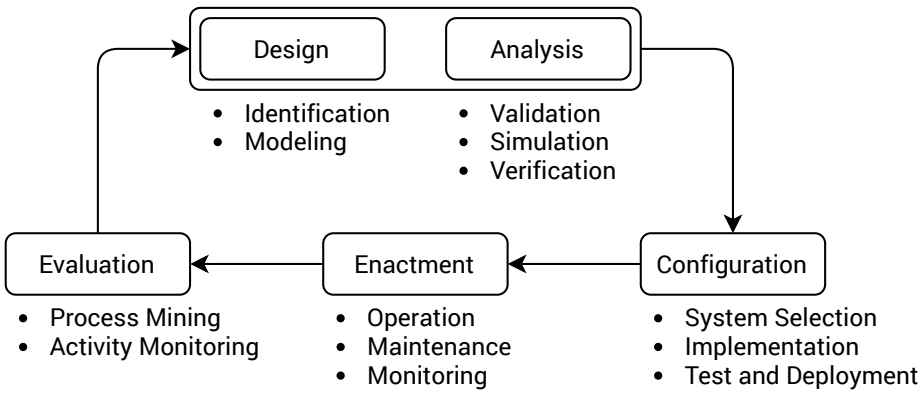


Figure 2.8: Business process lifecycle as adapted from Weske [112]

code, and testing the smooth operation of the result. Once the configuration completes, the process enters the *enactment* phase, in which it is operated in production. Carefully monitored and maintained, the process may be partly automated and executed.

Data from the enactment phase and feedback from persons involved then flows into an *evaluation* phase. Process mining may be used to detect whether the process was actually followed or whether there were deviations to address in the next iteration of the lifecycle.

2.3.2 Business Process Management Systems

Software systems can support organizations in implementing the business process lifecycle and facilitate their adoption of BPM. The culmination of this tooling support is the BPMS, which has its roots in early workflow management systems:

Definition 16 *(Business Process Management System).* A Business Process Management System (BPMS) is a software system that supports an organization in all phases of the business process lifecycle. ◇

In a recent systematic literature review, Pourmirza et al. analyzed 41 proposals for BPMS architectures, and found that there are fundamental differences in their scope, design, and level of abstraction. Particularly, there are multiple reference architectures and no clear preferences when it comes to adoption. Still, more than a third of the reviewed proposals (15) were at least partly based on the seminal Workflow Management Coalition (WfMC) reference model. This reference model also inspired the component-based BPMS architecture of Dumas et al. [31], which we will adopt as a reference in this thesis.

An overview of the architecture is shown in Fig. 2.9. In this context it is important to appreciate that a BPMS is assumed to run within the confines of a single organization, powering their internal processes. The central component is the *execution engine*, which drives the execution of pro-

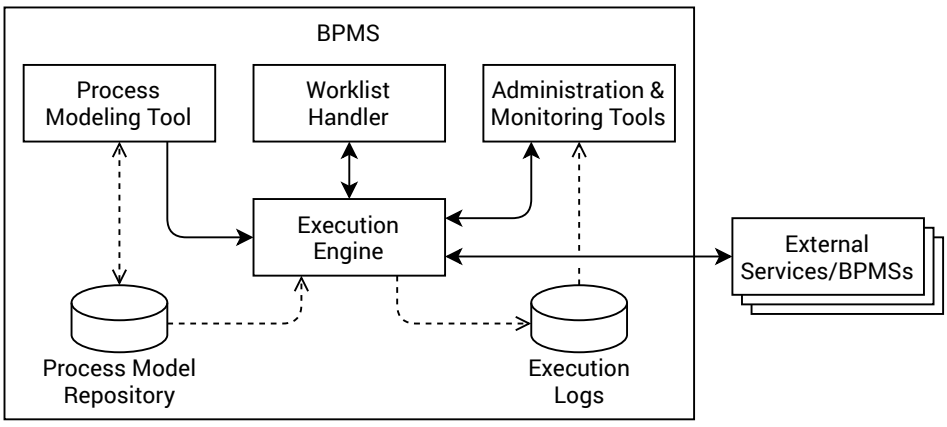


Figure 2.9: Architecture of a BPMS, adapted from Dumas et al. [31]

cesses based on process models. It is responsible for deploying processes, maintaining process data, and distributing work items to automated systems or human actors within the organization. The latter human actors interact primarily with the *worklist handler*, which offers interfaces to view assigned tasks, commit to them, and input data.

Process models are initially created using a *process modeling tool*, often employing visual languages like BPMN. The resulting models are stored in the *process model repository*, which can be queried by the execution engine during process execution. All runtime information and events occurring are also logged in *execution logs*, which are used by *administration and monitoring tools* to allow insights into the current process landscape.

Lastly, a BPMS—or groups of BPMSs partaking in a choreography—usually does not operate in an isolated environment, but interacts with other *external services* or BPMSs. External services may be queried for data interactions [94], for example. The connection to other BPMSs mainly serves the realization of business process choreographies.

2.3.3 Business Process Choreographies

Organizations may decide to collaborate with other organizations to optimize their operations, outsource specific tasks, and make use of business synergies. In BPM, these kinds of collaborations between multiple participants are captured using the notion of choreographies, and are described by choreography models. While ultimately realized via the participants' local process orchestrations, choreography models take on a more global view [112]:

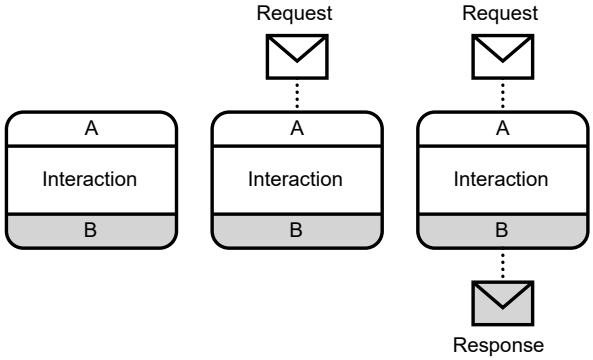
Definition 17 *(Choreography Model).* A choreography model specifies the interactions, mainly consisting of exchanges of control and data, that a number of participants agree upon to reach a common business goal. The choreography model specifies the ordering of said interactions and all associated constraints on a global level, essentially prescribing interaction interfaces for each participant. ◇

A choreography model may be described "as a business contract" [87] since it essentially contains terms and conditions of a collaboration agreement, and may thus pass through several negotiation and design phases as well [112]. It is the responsibility of the individual participants to subsequently design their business processes in a way that is compatible with the choreography model, meaning that the choreography directly affects the entire business process lifecycle.

Business Process Model and Notation

The de-facto standard modeling framework used in BPM is Business Process Model and Notation (BPMN), maintained by the Object Management

Table 2.1: BPMN choreography diagram notation (excerpt)

Choreography tasks		
		
Gateways		*Events*
	Parallel gateway	 (None) start event
	Data-based exclusive gateway	 Timer intermediate catch event
	Event-based exclusive gateway	 Conditional intermediate catch event
		 (None) end event
		 Terminate end event

Group (OMG), which provides both a metamodel as well as several diagram types for visualizations on different layers of abstraction, including choreographies [87].

A subset of the modeling elements available in choreography diagrams is shown in Tab. 2.1. The general structure is that flow nodes are connected using sequence flows, i.e., directed arrows, which impose a causal ordering. The main flow nodes are choreography activities, of which we only consider choreography tasks in this thesis: Each choreography task models an atomic message exchange between the initiator (the participant with the white background) and the receiver (the participant with the gray background). Optionally, message decorators may be used, and a response message may be specified to form a request-response exchange. The remaining choreography activities are structural in nature, and allow for hierarchical diagrams.

The other flow nodes used are gateways and events. Gateways are used to represent workflow patterns in which the control flow is somehow influenced. The parallel gateway splits the execution into parallel branches when used in a diverging direction, and joins them again when

used in a converging direction. The exclusive gateway similarly forks and merges execution branches based on an exclusive choice, so only one of the branches will be chosen. The event-based gateway is only used in a diverging direction, and represents the deferred choice pattern, an exclusive choice based on the occurrence of events [96].

The term "event" is somewhat overloaded in BPMN choreography diagrams and carries several meanings. For one, events mark the start of the choreography. They may also constrain the control flow by waiting for conditions on message data (conditional event) or the system time (timer event) to become true. Lastly, events mark the end of a specific execution branch (end event) or the choreography as a whole (terminate end event). In case of a termination, it must be assured that all participants are aware of the termination by being included in some previous message exchange.

Choreography models are usually used for documentation purposes. The main reason for this is the lack of a central coordinator who could enforce the choreography [87]. Instead, a choreography is realized through the business processes of the individual participants, which have to be compatible to the choreography model. That is, choreography models are not directly executed, but serve as a guideline for the underlying business processes of the participants [70]. The BPMN standard accordingly defines the semantics of choreography models in terms of a mapping to collaboration models, which consist of multiple process models and their interactions.

Chapter 3

Smart Contract Modeling

Proponents of smart contracts are largely unanimous in that their practical adoption depends—among other factors—on the availability of proper expression and modeling languages [111, 24]. The smart contract specifications ought to be understandable by all signatories and contain sufficient instructions for their partial or full enforcement via code [103]. Yet, comprehensively expressing smart contracts in such a way remains an unsolved challenge, and many approaches are only currently being explored [44]. Still, it is vital to have a common notion of what exactly constitutes a smart contract to argue about a management system or eventual enforceability concerns regarding DApps.

For the purpose of this thesis, we resort to a Model-Driven Engineering (MDE) approach [99]. To this end, we first clarify the taxonomy of smart contracts, and then reason towards a unifying metamodel of smart contracts (see Sect. 3.1). We give an example of how a smart contract model directly derived from the metamodel looks like, and introduce an operational smart contract semantics (see Sect. 3.2). We then apply the metamodel to an established modeling language from the BPM domain, BPMN choreography diagrams, and gauge whether they may be suitable to partially or fully specify smart contracts (see Sect. 3.3).

Parts of this chapter are based on various of our previous publications, in which we (i) introduced a unifying metamodel of legal smart contracts [66], (ii) discussed the appropriateness of BPMN choreography diagrams for smart contracts from an operational perspective [70], and (iii) provided a novel legal interpretation of these diagrams [65].

3.1 Smart Contract Metamodel

The goal of a metamodel in MDE is to "define the relationships among concepts in a domain and precisely specify the key semantics and con-

straints associated with these domain concepts" [99], which we will do in this section for the notion of smart contracts.

3.1.1 Terminology

The MDE approach is inherently hierarchical, as is evident from standards like the OMG Meta-Object Facility (MOF) [88], which has four layers from instance to meta-metamodel. It is helpful to clearly settle the terminology on every layer of the hierarchy (see Fig. 3.1): A *smart contract* is a concrete contractual agreement that is being performed by a number of parties engaging in legal relationships and fulfilling their agreed-upon obligations. The correct progression of the smart contract is enforced using two means: via tamper-proof execution of code, as well as via legal enforcement per the law. This understanding directly follows the definition of smart contracts from literature as presented in Sect. 2.1.3.

A smart contract is an instance of a *smart contract model*, which provides a general blueprint of the terms that were negotiated by the involved parties. Essentially, such a smart contract model is similar to a contract document in traditional legal contracts, which can be replicated arbitrarily often and is then signed to form a legal contract.

The smart contract model in turn is an instance of the *smart contract metamodel*. The metamodel describes and interrelates the fundamental concepts that may be contained in a smart contract model in an abstract fashion.

Finally, some researchers such as Hazard and Haapio advocate for a visual approach to smart contract modeling, referring to a general trend in law to augment legal prose with figures and clearer structuring [44].

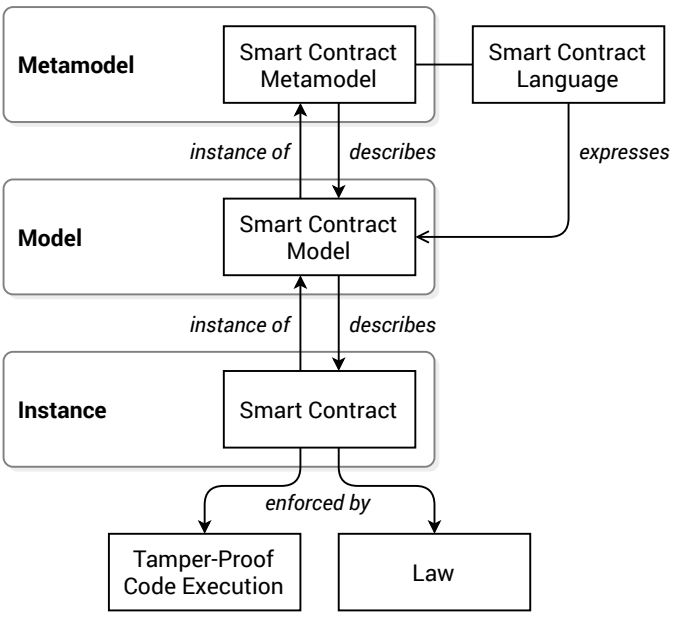


Figure 3.1: Overview of the MDE approach for modeling smart contracts

Metamodels define an abstract syntax, though, and the derived models are not inherently designed to be visually appealing or even a primary communication artifact between human stakeholders. For that, *smart contract languages* are used, which express smart contract models in different ways, including textual representations, for various purposes.

In the remainder of this chapter, we propose a generic smart contract metamodel encapsulating the current state-of-the-art understanding of smart contracts. It is not the goal, however, to conceive of a novel smart contract language, i.e., a DSL—instead, we will directly express smart contract models in terms of their abstract syntax. We will evaluate whether BPMN choreography diagrams may be an appropriate visual notation at the end of this chapter.

3.1.2 Reasoning

The early model of the Ricardian triple divided smart contracts into three distinct components: parameters, as well as operational (code) and non-operational (legal prose) aspects (see Sect. 2.1.3). This understanding persists in state-of-the-art attempts to implement smart contracts like Corda, in which a smart contract's non-operational terms and conditions are attached to an operational implementation as static data which is not necessarily machine-readable [45].

While the presence of such non-operational aspects places a hard limit on the potential for automation, it is widely considered to be a necessity given the intricacy of the legal domain [24]. Even more so, many contracts may be deliberately designed to be ambiguous or leave room for interpretation for reasons such as flexibility and adaptability [39]. We thus adopt the general distinction introduced in the Ricardian triple for our metamodel as well, but strive for a high degree of coupling between operational and non-operational aspects to close the gap between them.

General Structure

The operational side pertains to the behavior of the parties and the smart contracts itself. Behavior in modeling is typically expressed using behavioral models such as Petri nets [93]. In fact, Petri nets were very early on discovered as an adequate formalism for the operational side of contracts in the seminal work on electronic contracting by Lee [73]. There, the authors employ Petri nets to argue about causal and relative temporal relationships through the structure of the net in addition to first-order logic specifications.

In general, a Petri net is a directed graph containing two types of nodes, places and transitions, which are connected via flows [93]. Places can hold tokens, which are consumed and produced by transitions as specified by the incoming and outgoing flows, respectively. This relatively simple model allows the specification of complex workflow patterns like par-

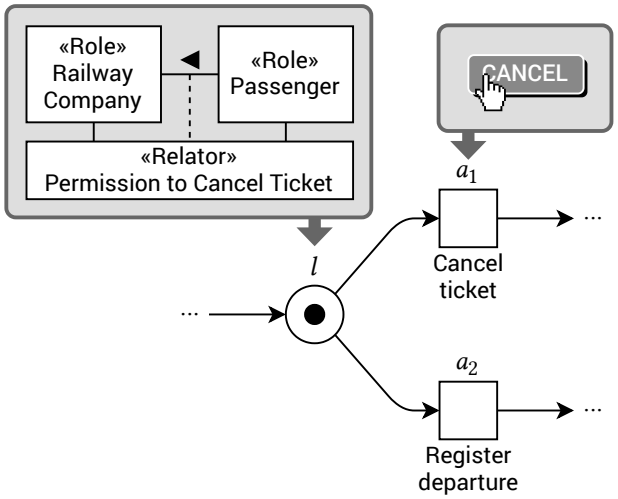


Figure 3.2: Petri net model of a legal relations and associated actions in the train ticket scenario

allel split, exclusive choice, or deferred choice [96]. The number and location of the tokens within the Petri net constitutes its state.

We adopt Petri nets as the basis for our smart contract metamodel following the proposal by Lee [73]. The idea is that places capture the legal relations between the participants. A place could, for example, be connected to a complex legal relation modeled as an UFO-L instance, capturing also non-operational contract aspects. A token on the place signifies that the associated legal relation currently holds, and at the same time enables certain actions—that is, transitions—to be performed. This results in an interplay of operational and non-operational aspects, which can be specified in different levels of detail.

Figure 3.2 shows an example of this understanding of a mutual interplay between actions and legal relations using Petri nets on an excerpt from the train ticket scenario. The place l represents a legal relation between the railway company and the passenger, e.g., the right to use a specific train as modeled using a simplified UFO-L model [40]. This enables two actions: a_1 to cancel the ticket, and a_2 which registers the departure of the train and validates the ticket.

Requirements

Of course, smart contracts do not only rely on control flow constructs and may require more expressive modeling elements. While general modeling formalisms like colored Petri nets [54] could be used to add capabilities, we strive for a more domain-specific and deliberate extension based on concrete requirements for (smart) contract specifications.

We primarily base our requirements on a survey by Hvitved, who identified 16 such requirements from an "ought-to-be" perspective [50]. That is, contract formalisms were compared with the intention that they should specify what should be done, not what the current state of affairs is—which

Table 3.1: Requirements for contract formalisms by Hvitved [50]

No	*Description*	*Metamodel*
R_1	Contract model, language, semantics	—
R_2	Contract participants	✓
R_3	(Conditional) commitments	✓
R_4	Absolute temporal constraints	✓
R_5	Relative temporal constraints	✓
R_6	Reparation clauses	(✓)
R_7	Instantaneous and continuous actions	✓
R_8	Potentially infinite and repetitive contracts	✓
R_9	Time-varying, external dependencies (observables)	✓
R_{10}	History-sensitive commitments	✓
R_{11}	In-place expressions	✓
R_{12}	Parametrized contracts	✓
R_{13}	Isomorphic encoding	✓
R_{14}	Run-time monitoring	—
R_{15}	Blame assignment	—
R_{16}	Amenability to (compositional) analysis	—

makes the requirements useful for capturing the operational aspects of contracts. The requirements are reproduced in Tab. 3.1.

With the exception of R_1, a meta-requirement, and R_{14}–R_{16}, which describe particular use-cases, all requirements pertain to structural and operational features and will be represented in our metamodel in some form as indicated in Tab. 3.1. We will refer to these requirements throughout this chapter.

Pace and Schneider examine further challenges and associated requirements, in particular concentrating on temporal and causal relationships between deontic notions, most of which are already covered above [89]. However, they additionally refer to *introspection* and *reflection*, i.e., the capability of a smart contract model to refer to itself or parts of itself and its current state. The operational semantics we will introduce later in this thesis support this to a degree by allowing conditions to reference the entire history of an instance.

As a preliminary disclaimer, we will not consider separate reparation clauses (R_6) or exception clauses [89] in the metamodel, since they can principally be modeled in the same way as the "happy path" and do not require special treatment from a metamodel perspective. Smart contract languages may still be designed to provide elements specifically targeting exceptions.

3.1.3 Metamodel Structure

Taking into account the reasoning outlined above, we compiled a comprehensive metamodel for smart contracts based on the general structure of

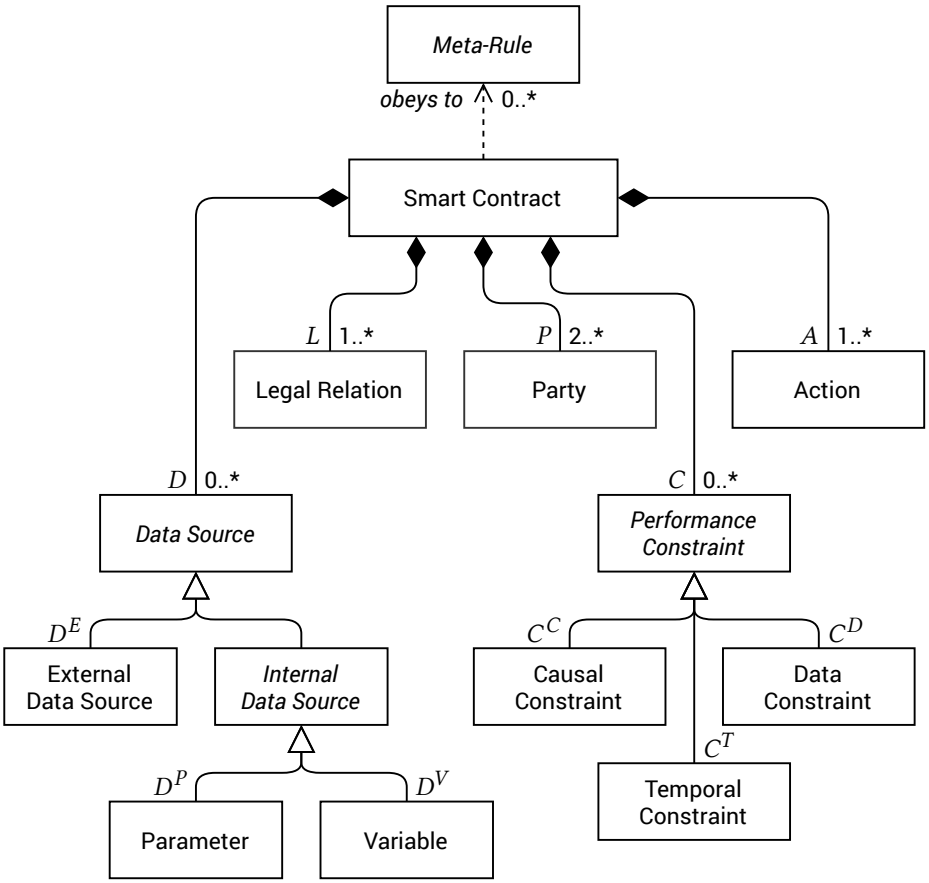


Figure 3.3: Containment and inheritance hierarchy of the smart contract metamodel with some associations omitted for brevity

Petri nets. An abbreviated version is shown in Fig. 3.3, which gives rise to the following formal notion of a smart contract model:

Definition 18 *(Smart Contract Model).* A smart contract model $\mathcal{M} = (P, D, L, A, C)$ is a tuple containing

- P a set of at least two parties,
- D a set of data sources, subdivided into
  - D^E external data sources,
  - D^P internal parameters,
  - D^V internal variables,
- L a non-empty set of legal relations between the parties,
- A a non-empty set of actions, and
- C a set of performance constraints, subdivided into
  - C^C causal constraints,
  - C^T temporal constraints, and
  - C^D data constraints. ◇

Smart contracts are subject to the laws, norms, and restrictions within the jurisdiction or jurisdictions they are used in. Meta-rules (see Fig. 3.3) describe those additional elements of a smart contract which are not explicitly part of the smart contract model itself [59]. Instead, references to legal texts or bills would be embedded in the smart contract model, referencing relevant sections of law. We do not impose any specific way to represent these references.

Parties and Legal Relations

A smart contract model $\mathcal{M} = (P, D, L, A, C)$ represents the blueprint of an agreement between at least two parties P who are eventual signatories to the smart contract, akin to the notion of smart contract templates [24]. A party may be a person, an organization, or any other entity capable of entering into a legal agreement [37]. In the train ticket scenario, for instance, the set of parties could be $P = \{\mathrm{PS}, \mathrm{RC}, \mathrm{PP}\}$, meaning the passenger, railway company, and payment provider, respectively. They are ultimately responsible for performing actions for the duration of the smart contract.

The parties P enter into a contractual agreement, and are subject to legal relations (see Sect. 2.1.2):

Definition 19 *(Legal Relation).* Let $\mathcal{M} = (P, D, L, A, C)$ be a smart contract model. Then a legal relation $l \in L$ relates a subset of parties of a smart contract model $\mathcal{M}$. A legal relation is further specified by a predicate $\mathsf{Initial} : L \longrightarrow \{\mathsf{true}, \mathsf{false}\}$, which is true iff the legal relation holds immediately in every new smart contract instance. ◇

Concrete legal relations could be expressed using the fundamental legal relations of Hohfeld [48] including obligations and permissions, complex ontology models like UFO-L [40], or any other formalism capable of relating parties.

In the course of this thesis, we will stick to the former fundamental legal relations of Hohfeld on account of their conciseness. For example, an obligation $l := \text{Obligation}_{\text{PS,PP}}(\textit{Pay upgrade fee})$ then is a legal relation, stating that the passenger has the obligation to pay a fee via the payment provider. Legal relations could also be more complex or hierarchical, and even relate more than two parties.

Data Sources

The data sources D are crucial to the functioning of a smart contract: *Parameters* D^P represent the individual results of the negotiation phase for a particular smart contract instance. For example, the price of a train ticket may vary depending on which store the passenger bought it at, or which kind of discount card they possess. Parameters are an inherent part of the smart contract from its instantiation, and basically allow template-like modifications of each new instance.

Variables D^V, on the other hand, are things the contract keeps track of itself, e.g., a total of already paid expenses or whether a class upgrade was purchased by the passenger. They are similar to parameters, but are not negotiated beforehand and may be set and changed during runtime of a smart contract in updates due to an action. Parameters and variables are internal data sources, meaning they are stored within and at the responsibility of the smart contract.

This is not the case for all data sources, though. Contracts often make use of external data, as expressed by *external data sources* D^E. For example, the price of a stock from the stock exchange or the current weather warning level according to a meteorological service may be used within smart contract models, without its managing entity being a legally bound party in the smart contract. In Tab. 3.1, this is called a "time-varying, external observable" (R_9) [50]. When using DApps as an enforcement mechanism, those external data sources would be provided by oracle services (see Chapter 4).

We will largely abstract from the types, formats, and individual domains of data sources, instead using a generic common data domain for all of them in the following:

Definition 20 *(Data Domain).* The data domain $\mathbb{D}$ contains all possible values of data. ◇

Actions

Actions (R_7) are the center pieces of smart contract behavior:

Definition 21 *(Actions).* Let $\mathcal{M} = (P, D, L, A, C)$ be a smart contract model. Then an action $a \in A$ encapsulates activities, operations, or other behavior which may be performed, and which is atomic from an enforcement point of view. Actions are further specified by

- $\mathsf{Pre} : A \longrightarrow \mathcal{P}(L)$, a set of legal relations the action requires to hold and subsequently extinguishes,
- $\mathsf{Post} : A \longrightarrow \mathcal{P}(L)$, a set of legal relations created or otherwise caused by performing the action,
- $\mathsf{ChoiceOf} : A \longrightarrow \mathcal{P}(P)$, a set of parties who have the choice and subsequently share the responsibility of performing the action,
- $\mathsf{Terminating} : A \longrightarrow \{\mathsf{true}, \mathsf{false}\}$, a predicate indicating whether performing this action terminates the smart contract. ◇

Actions may also read from data sources and write to variables, although this will not be further specified in the formalization.

Our smart contract models follow a structure closely related to Petri nets, and actions can be seen as the equivalent to transitions: They consume or extinguish a number of legal relations (the preset Pre of the transition), and create new ones (the postset Post of the transition). In conjunction with the legal relations L, this interlinking essentially provides an isomorphic mapping (R_{13} in Tab. 3.1) between non-operational and operational contract aspects, or legal relations and actions, respectively.

The decision whether an action $a \in A$ is actually performed is the choice of one or a group of parties $\mathsf{ChoiceOf}(a)$. If there is more than one party assigned to an action in this way, they must reach consensus on whether to perform the action. Note that performing actions can also be delegated within parties. For example, an organization has many employees capable of acting in the name of the organization, and some actions be even be automated by software systems. However, this kind of automation is done within within an organization, that is, it pertains to the internal business logic and deliberation of a party.

There is a second kind of automation in smart contracts, of those actions which are completely autonomous and disassociated from any single party:

Definition 22 *(Autonomous Actions).* Let $\mathcal{M} = (P, D, L, A, C)$ be a smart contract model. Then an action $a \in A$ is *autonomous* iff no party is responsible for its performance, i.e., $\mathsf{ChoiceOf}(a) = \emptyset$. Autonomous actions are enforced automatically via tamper-proof code execution. ◇

Autonomous actions and their enforcement are the fundamental novelty of smart contracts as compared to their traditional legal contract counterpart. Implicitly, they elevate the smart contract itself to be an active party, although narrowly bound by the specification of the model. No party

has any direct power over an autonomous action, since it is executed via tamper-proof code. In the train ticket scenario, for instance, the automatic validation of the ticket as soon as the booked train departs is an example of an autonomous action.

In the metamodel we do not impose any level of abstraction for the specification of actions, including how exactly performance constraints or modifications of data sources are expressed. Actions could, for example, contain many steps—similar to a process—with multiple involved actors. In that case, an implementation of the smart contract must make sure that proper locking and concurrency mechanisms are used so actions do not interfere with each other. In the following, we will assume for the sake of simplicity that actions are internally atomic and can be performed instantaneously.

Performance Constraints

Whether an action can be performed or not may be subject to so-called performance constraints. They model terms and condition of the contractual agreement which pose restrictions as to when an action may be performed, and pertain to several contract requirements (R_3–R_5, R_{10}, R_{11} in Tab. 3.1).

Definition 23 *(Performance Constraints).* Given a smart contract model $\mathcal{M} = (P, D, L, A, C)$, a performance constraint $c \in C$ ties the performance of an action to additional conditions being satisfied. A performance constraint affects one or more actions $\mathsf{Affects} : C \longrightarrow \mathcal{P}(A)$. Further, a data performance constraint reads some data sources $\mathsf{Reads} : C^D \longrightarrow \mathcal{P}(D)$. ⋄

Causal constraints express requirements regarding past actions. This is mostly used to model immediate consecutiveness of actions, e.g., payment before delivery, or exclusive and parallel branching. It should be noted that some types of causal constraints may also be modeled using legal relations and the Pre and Post sets. However, this may require duplication of equivalent legal relations to accommodate for actions $a \in A$ with $\mathsf{Pre}(a) = \mathsf{Post}(a)$ that are still part of a causal chain of actions. This could potentially pollute the smart contract model, especially when more complex patterns are concerned. We thus opted for the separate specification of complex causal constraints.

Temporal constraints describe situations in which an action may require waiting until a certain deadline has passed. Similar mechanisms to

Table 3.2: Temporal constraint definitions in ISO-8601 format

Name	*Example*	
Date	`2020-12-24T12:00:00Z`	(noon on Dec. 24th, 2020, UTC)
Duration	`P7D`	(7 days)

timer events in process and choreography models may be obtained [33, 20], including relative and absolute timers. Some examples of concrete specifications are given in Tab. 3.2: *dates* are used to wait for a specific, absolute point in time; *durations* are used to delay the smart contract execution or represent timeouts. For this, all parties must have a common understanding about the current time, and what time means in the implementation environment of the smart contract. We will discuss the challenges in this in more detail in Sect. 3.2.3 and Chapter 7.

Lastly, *data constraints* make sure that the values of the data sources fulfill some conditions. For example, an action to perform a stock option may be contingent on the price of the stock exceeding a predefined strike price. Alternatively, an obligation to buy additional insurance for a delivery might depend on the value of the ordered products. Data constraints are usually specified in the form of expressions, which can be evaluated by a software system. In Chapter 6, we will discuss the semantics of data constraints in detail, and for now continue with the operational semantics of smart contracts as a whole.

3.2 Operational Semantics

Given a fully specified smart contract model $\mathcal{M} = (P, D, L, A, C)$, it can be instantiated to yield arbitrarily many smart contract instances $\mathcal{S}_1$, $\mathcal{S}_2$, and so on. These instances are executed according to the operational semantics we are going to introduce in this section.

3.2.1 Running Example

In the remainder of this thesis, we will consider a concrete smart contract model $\mathcal{M}_{ticket} = (P, D, L, A, C)$ of a contract of carriage—that is, a train ticket—as a running example.

There are three parties, $P = \{\mathrm{PS}, \mathrm{RC}, \mathrm{PP}\}$, meaning the passenger, railway company, and payment provider, respectively. The passenger wishes to take a train to their destination, and for that reason buys a train ticket. The negotiation phase is not part of the smart contract; in our case, the passenger picked a second class ticket, prohibiting them from entering the first class area of the train. They also made use of a personal subscription-based discount card granting them a percentage off the ticket price.

Figure 3.4 shows a Petri net visualization of the actions $A = \{a_1, ..., a_{10}\}$ and legal relations $L = \{l_1, ..., l_6\}$ as transitions and places, respectively, with the pre and post sets indicated by flows. The initial legal relations with $\mathsf{Initial}(l) = \mathsf{true}$ and terminating actions with $\mathsf{Terminating}(a) = \mathsf{true}$—those immediately ending the smart contract once performed—are shaded with a gray background, respectively. Further, autonomous actions are outlined with a bold stroke, and the set $\mathsf{ChoiceOf}(a)$ of parties in charge of an action a is added to its label in parentheses. Additional information—

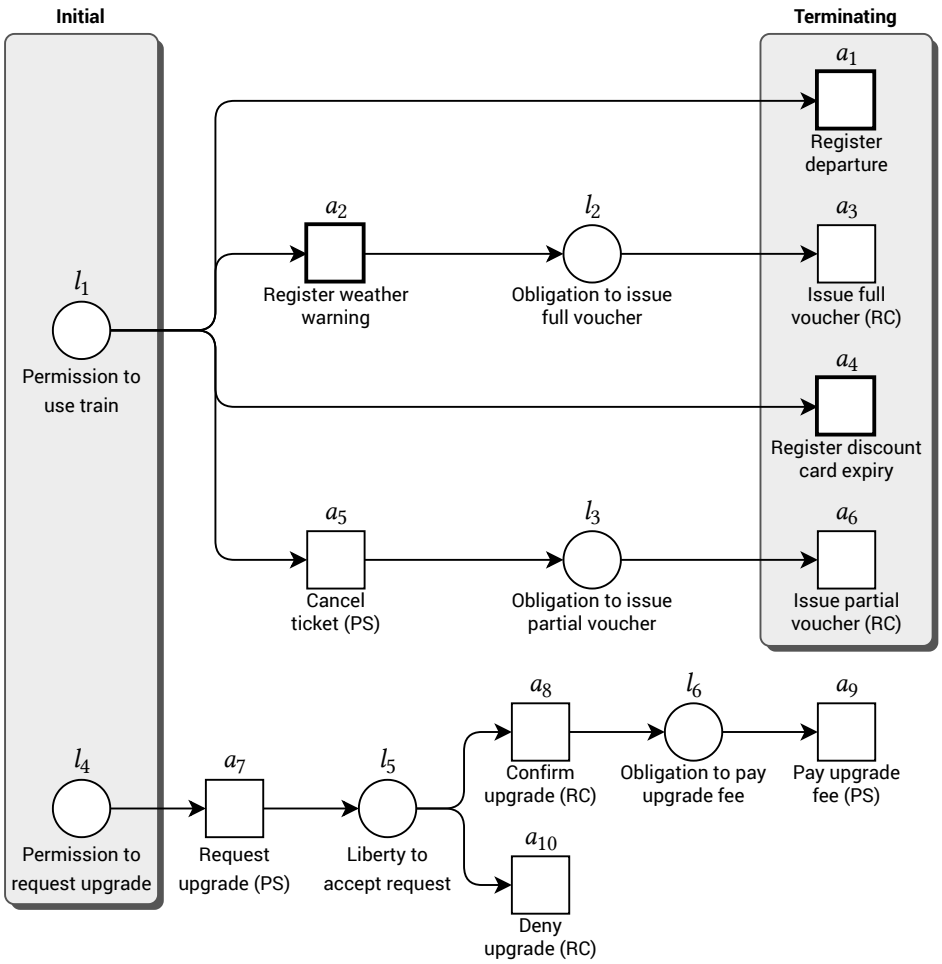


Figure 3.4: Actions and legal relations of $\mathcal{M}_{ticket}$

Table 3.3: Additional specification of the smart contract $\mathcal{M}_{ticket}$

Data sources		
$D^E = \{d_w, d_d\}$	d_w : Weather warning level	
	d_d : Train departure status	
$D^P = \{d_p, d_c\}$	d_p : Ticket price	
	d_c : Ticket class	
$D^V = \{d_u\}$	d_u : Upgrade indicator	
Performance constraints		
$C^C = \emptyset$		
$C^T = \{c_c\}$	c_c : Discount card is expired	$\mathsf{Affects}(c_c) = \{a_4\}$
$C^D = \{c_d, c_w\}$	c_d : Train has departed	$\mathsf{Reads}(c_d) = \{d_d\}$
		$\mathsf{Affects}(c_d) = \{a_1\}$
	c_w : Severe weather warning is active	$\mathsf{Reads}(c_w) = \{d_w\}$
		$\mathsf{Affects}(c_w) = \{a_2\}$
Legal relations		
l_1 : $\mathsf{Permission}_{\mathrm{PS,RC}}$(*Use train subject to cancelations*) [$\mathsf{Initial}(l_1) = \mathsf{true}$]		
l_2 : $\mathsf{Obligation}_{\mathrm{RC,PS}}$(*Issue full voucher*)		
l_3 : $\mathsf{Obligation}_{\mathrm{RC,PS}}$(*Issue partial voucher*)		
l_4 : $\mathsf{Permission}_{\mathrm{PS,RC}}$(*Request upgrade*) [$\mathsf{Initial}(l_4) = \mathsf{true}$]		
l_5 : $\mathsf{Liberty}_{\mathrm{RC,PS}}$(*Accept upgrade request*)		
l_6 : $\mathsf{Obligation}_{\mathrm{PS,PP}}$(*Pay upgrade fee*)		

that is, the performance constraints, data sources, and full form of the legal relations—is given in Tab. 3.3.

The smart contract has a simple "happy path": The passenger makes their way to the train station and takes the train upon departure. The ticket is automatically validated as soon as the train departs (a_1) and the smart contract terminates. This is contingent on the performance constraint c_d, which reads the external data source d_d holding the train's departure status. The departure status is managed and maintained by the Railway Infrastructure Manager (RIM), which is not explicitly modeled.

Some things may happen before, though. For one, a severe weather warning may be issued by the meteorological service as modeled by the external data source d_w, which satisfies the data performance constraint c_w triggering the autonomous action a_2. This cancels the ticket for precautionary safety reasons, but creates the obligation l_2 for the railway company to issue a full voucher (a_3). Similarly, the passenger's discount card may expire, satisfying the temporal performance constraint c_c and triggering the terminating autonomous action a_4—that is, there is no refund or voucher. Lastly, the passenger may themselves decide to cancel the ticket (a_5), after which they have the right (l_3) to receive a partial voucher (a_6).

In parallel, the passenger may also decide to request and upgrade to first class (a_7), which is usually a possibility that is intended and written in the smart contract (l_4) but entirely at the liberty of the railway company

(l_5). Of course, this requires an additional payment (l_6) in coordination with the payment provider (a_9), who is a third party to the contract.

Note that this example is chosen to be simple, and practical smart contracts may be much larger. We also deliberately do not fully specify all concepts; for example, actions read from and write to internal data sources. In the scope of this thesis, these relationships are not explored further, but the metamodel and the following operational semantics do not principally restrict their usage either.

3.2.2 State Space

An instance $\mathcal{S}$ of a smart contract model $\mathcal{M} = (P, D, L, A, C)$ is primarily characterized by its single state at any point in time. The state contains all information specific to the runtime of $\mathcal{S}$, particularly including the valuation of internal data sources:

Definition 24 *(Smart Contract State).* Let $\mathcal{M} = (P, D, L, A, C)$ be a smart contract model. Then $\mathbb{S}^{\mathcal{M}}$ is the set of all states of instances of $\mathcal{M}$, and a state $s = (\Lambda, v, \mathrm{H}) \in \mathbb{S}^{\mathcal{M}}$ is a tuple with

- $\Lambda \subseteq L$ a subset of the legal relations which hold in the state,
- $v : D^P \cup D^V \longrightarrow \mathbb{D}$ a valuation function assigning all internal data sources to a value from the data domain $\mathbb{D}$, and
- H a full log of all actions, their effects, and involved parties since the smart contract was first instantiated. ◇

We will usually refer to the domain of states $\mathbb{S}$ without the superscript annotation, if the smart contract model it applies to is clear from the context.

The smart contract metamodel is based on Petri nets, and Λ corresponds to the marking of the places with tokens [93], in particular resembling a 1-bounded condition event net [112, Sect. 4.2.1]. The log H ensures a complete and consistent history of everything that has happened during the execution of a smart contract. This information may be used for monitoring or blame assignment (R_{14} and R_{15} in Tab. 3.1). However, it is also used by performance constraints to check whether their specific conditions are satisfied, which often depends on past actions like in the case of causal constraints. Concrete implementations will specify a sensible data structure to store and retain only explicitly required information.

In any case, when a smart contract $\mathcal{S}$ is initially instantiated, it takes on an initial state:

Definition 25 *(Initial Smart Contract States).* Let $\mathcal{M} = (P, D, L, A, C)$ be a smart contract model. Then a state $s = (\Lambda, v, \mathrm{H})$ is in the set of initial states $\mathbb{S}_0 \subseteq \mathbb{S}$ iff $\Lambda = \{l \in L \mid \mathsf{Initial}(l) = \mathsf{true}\}$ and the log H is empty. ◇

Initial states reflect the results of case-by-case negotiations. For example, instances of the same contract model could start with different valuations of the parameters D^P, like different ticket prices offered by different vendors. Lastly, once a smart contract has finished, it reaches a final state characterized by the absence of any currently holding legal relations:

Definition 26 *(Final Smart Contract States).* Let $\mathcal{M} = (P, D, L, A, C)$ be a smart contract model. Then a state $s = (\Lambda, \nu, \mathrm{H})$ is in the set of final states $\mathbb{S}_F \subseteq \mathbb{S}$ iff $\Lambda = \emptyset$, that is, no legal relations hold anymore. ◇

3.2.3 Operating Environment

While a smart contract state $s = (\Lambda, \nu, \mathrm{H})$ contains all information a smart contract $\mathcal{S}$ is responsible for itself, it can never be viewed in complete isolation. Indeed, it is crucial to consider the environment in which $\mathcal{S}$ operates. This includes the external data sources D^E of the smart contract model $\mathcal{M}$, which are maintained by some entity independent of the smart contract. We define the state of the operating environment as follows:

Definition 27 *(Operating Environment States).* Let $\mathcal{M} = (P, D, L, A, C)$ be a smart contract model. Then $\mathbb{O}^{\mathcal{M}}$ is the set of all operating environment states relative to $\mathcal{M}$, and a state $o = (t, \phi) \in \mathbb{O}^{\mathcal{M}}$ is a tuple with

- $t \in \mathbb{N}$ the current system time, and
- $\phi : D^E \longrightarrow \mathbb{D}$ a valuation function assigning a value to each external data source.

Two operating environment states $(t, \phi), (t', \phi') \in \mathbb{O}^{\mathcal{M}}$ directly follow each other, or $(t, \phi) \longrightarrow (t', \phi')$, iff $t + 1 = t'$. ◇

Again, if the smart contract model $\mathcal{M}$ is clear from the context, we will omit the superscript annotation on the state domain.

Apart from the valuation of the external data sources, an operating environment state also contains a notion of a *system time*. This is the normative time that all parties agree to adhere to, for example Coordinated Universal Time (UTC) or Unix time. The current timestamp could either be provided by a time server, some form of consensus between the parties, or a blockchain network. We will discuss the challenges involved in finding such an agreed-upon time in Chapter 7.

Operating environment states thus describe the passing of time in the "real world". At each point in time, there is exactly one valid and unique operating environment state, which then transitions to the next one via the directly-follows relationship $\longrightarrow$. The frequency of such transitions is obviously limited by the general computing infrastructure which is used, but once per second or once per millisecond are common examples. For now, we assume that all parties and the smart contract itself have ready and consistent access to this operating environment state, and abstract from any network influences or latencies.

3.2.4 State Transitions

A smart contract instance $\mathcal{S}$ is in some state $s = (\Lambda, \nu, \mathrm{H}) \in \mathbb{S}$, until an action is performed and $\mathcal{S}$ transfers to some new state $s' \in \mathbb{S}$. In the following, we will describe state transitions such as this formally.

Enablement of Actions

An action can of course not be performed arbitrarily, but it must be enabled. Enablement is contingent on two factors: (i) the presence of all required legal relations $\mathsf{Pre}(a)$ in Λ, as well as (ii) the satisfaction of all attached performance constraints taking into account the current operating environment state $o = (t, \phi) \in \mathbb{O}$.

The performance constraints rely on various components of s and o. Causal constraints, for example, consult the log of actions H. Absolute temporal constraints check the system time t and see whether a deadline has been passed. Data constraints evaluate the valuation ν of internal data sources, as well as the valuation ϕ of external data sources. We summarize this using the following abstract definition:

Definition 28 *(Satisfaction of Performance Constraints).* Let $\mathcal{M} = (P, D, L, A, C)$ be a smart contract model. Then $\gamma : C \times \mathbb{S} \times \mathbb{O} \longrightarrow \{\mathsf{true}, \mathsf{false}\}$ is a Boolean predicate determining whether a performance constraint $c \in C$ is *satisfied* in a smart contract state $s \in \mathbb{S}$ and an operating environment state $o \in \mathbb{O}$. ◇

In the train ticket model $\mathcal{M}_{ticket}$, we could specify that $\gamma(c_c, s, o) \equiv t > 1617012992$, that is, the passenger's discount card expires when the system time passes 1617012992, which is the Unix time as of writing this sentence. Further, we could specify that $\gamma(c_w, s, o) \equiv \phi(d_w) \geq 2$, that is, a severe weather warning corresponds to a weather warning level d_w greater or equal to 2. Lastly, $\gamma(c_d, s, o) \equiv \phi(d_d) = \mathsf{departed}$ specifies that the train has officially departed once the external data source equals the literal string value departed.

Then it is straightforward to determine whether an action is enabled:

Definition 29 *(Action Enablement).* Let $\mathcal{M} = (P, D, L, A, C)$ be a smart contract model. Then, given a smart contract state $s = (\Lambda, \nu, \mathrm{H}) \in \mathbb{S}$ and an operating environment state $o = (t, \phi) \in \mathbb{O}$, an action $a \in A$ is

- enabled, if all legal relations a consumes hold in s and all performance constraints that affect a are satisfied,
- active, if all legal relations a consumes hold in s, but there are non-satisfied performance constraints, or
- disabled, if not all legal relations a consumes hold in s.

Formally, action enablement is described by the function $\sigma : A \times \mathbb{S} \times \mathbb{O} \longrightarrow \{\mathsf{disabled}, \mathsf{active}, \mathsf{enabled}\}$ with

$$\sigma(a, s, o) := \begin{cases} \mathsf{disabled} & \text{if} \quad \exists\, l \in \mathsf{Pre}(a) : l \notin \Lambda \\ \mathsf{active} & \text{if} \quad \mathsf{Pre}(a) \subseteq \Lambda \wedge \\ & \qquad \exists c \in C : \big(a \in \mathsf{Affects}(c) \wedge \neg\gamma(c, s, o)\big) \\ \mathsf{enabled} & \text{if} \quad \mathsf{Pre}(a) \subseteq \Lambda \wedge \\ & \qquad \forall c \in C : \big(a \in \mathsf{Affects}(c) \longrightarrow \gamma(c, s, o)\big) \end{cases}$$

◇

For example, consider an initial state $s = (\Lambda, \nu, \mathrm{H}) \in \mathbb{S}_0$ of the train ticket smart contract model $\mathcal{M}_{ticket}$ with $\Lambda = \{l_1, l_4\}$. Then $\sigma(a_7, s, o) = \mathsf{enabled}$ for any operating environment state o, since the passenger may request an upgrade without constraints. However, $\sigma(a_2, s, o) = \mathsf{active}$ only holds as long as the weather warning level stays below 2 in o, after which it becomes enabled.

Enforceability of Actions

An action that is enabled is *ready* to be performed; whether an action *must* be performed, however, is a question closely related to enforceability whose answer differs between autonomous and non-autonomous actions.

Autonomous actions are performed by the smart contract itself. Since the purpose of the tamper-proof smart contract enforcement is to alleviate the potential for conflicts and reduce any undue influence on the contractual proceedings, the parties may rightfully assume that there is no leeway when it comes to their performance. Instead, the smart contract itself should be entirely neutral, and should cause no delay potentially affecting any party. We thus require that autonomous actions are immediately performed without delay as soon as they become enabled. If delays of some sort are required, they can be modeled as a temporal performance constraint which can be autonomously checked.

Of course, multiple autonomous actions may become enabled at the same time. In that case, one is chosen non-deterministically, even if that leads to the second action becoming disabled again. Autonomous actions have priority over non-autonomous actions, that is, they are always performed first. This circumstance should be kept in mind when designing smart contract models to avoid any conflict of interest.

The enforcement of non-autonomous actions is complicated since it runs into a very practical issue: It is unethical and—from the perspective of the smart contract—impossible to physically force any party to perform an action, even if that action is borne by an obligation and they are responsible for it. This circumstance is the main reason why smart contracts still rely on a secondary enforcement mechanism via the law, if the behavior of individual parties makes it impossible for the tamper-proof code to proceed according to the smart contract model. In that case, arbitration or litigation proceedings have to kick in on the basis of the tamper-proof

smart contract history and log H which is maintained in the smart contract state.

In the train ticket example $\mathcal{M}_{ticket}$, for instance, there is no way for the smart contract to force the passenger to perform a_9 and pay the upgrade fee, and the smart contract may even terminate before by a terminating action being performed. While smart contracts can be designed to mitigate some of these effects, e.g., by explicitly containing penalties and clauses to pressure parties into proper cooperation, this forms a fundamental limit to the automated enforceability of smart contracts in general. In the scope of this thesis, we thus only consider the enforcement of smart contracts *before* the lawsuit.

Performance of Actions

Based on its own state and that of the operating environment, a smart contract $\mathcal{S}$ can now transition to a new state when an action is performed, subject to the restrictions introduced above. We formally define this as follows:

Definition 30 *(Smart Contract State Transitions).* Let $\mathcal{M} = (P, D, L, A, C)$ be a smart contract model, $s = (\Lambda, v, \mathrm{H}) \in \mathbb{S}$ be a state of the smart contract, and $o = (t, \phi) \in \mathbb{O}$ the current operating environment state.

Then an action $a \in A$ can be performed in o leading to a new state $s' = (\Lambda', v', \mathrm{H}') \in \mathbb{S}$, or

$$s \xrightarrow{o/a} s'$$

iff $\sigma(a, s, o) = \mathsf{enabled}$ and no autonomous action must be performed first. The new state s' is defined as follows:

(i) $$\Lambda' := \begin{cases} \emptyset & \text{if } \mathsf{Terminating}(a) \\ \big(\Lambda \setminus \mathsf{Pre}(a)\big) \cup \mathsf{Post}(a) & \text{otherwise} \end{cases}$$

(ii) The effects of a are applied to the set of internal data sources it modifies resulting in a new valuation function v'.

(iii) All relevant information about the action, current system time, and associated parties is appended to H'. ◇

In Fig. 3.5, we show an excerpt of the state space of an instance $\mathcal{S}$ of the example train ticket smart contract model $\mathcal{M}_{ticket}$ introduced in Sect. 3.2.1. The smart contract starts in a state $s_0 \in \mathbb{S}_0$. The top path models the passenger requesting an upgrade, performing a_7 at o_1 to reach state s_1. This request could be accepted by the railway company performing, for example, a_8 at o_4 to reach state s_3. It could also be rejected by performing a_{10} at o_3 to reach state s_2. In all cases, the train departs at o_7 as indicated by the train departure status in the external data source d_d changing to departed, which satisfies the performance constraint c_d independent of the smart

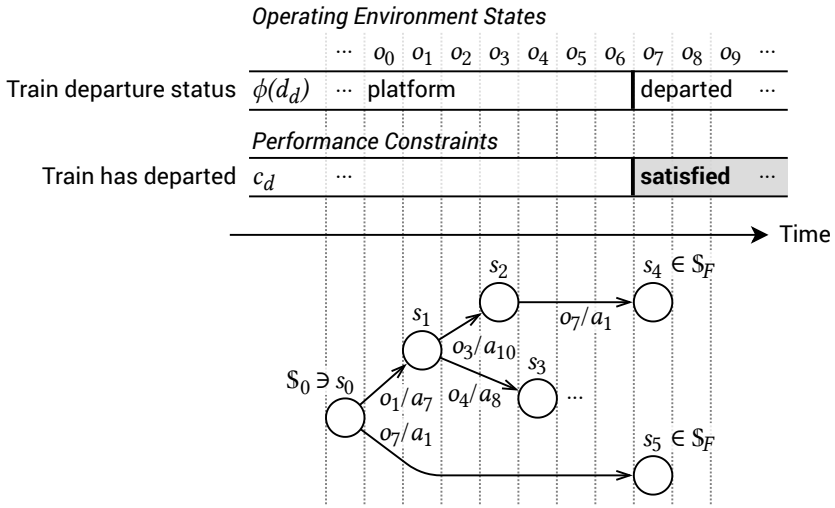


Figure 3.5: Excerpt of a state space of a smart contract

contract state. This in turn enables the autonomous action a_1 which registers the train's departure and validates the ticket, terminating the smart contract by reaching $s_4, s_5 \in \mathbb{S}_F$.

3.3 Choreographies and Smart Contracts

The smart contract metamodel allows us to reason about smart contracts and their components, to understand and define their semantics, and to eventually enforce these semantics in an actual execution environment. We did not, however, propose any non-mathematical notation for smart contracts (see Sect. 3.2.1), falling short of the domain's vision of providing an understandable representation for all parties and stakeholders [24].

It is not the goal of this thesis to develop such a notation. Yet, there are domains facing similar issues, chiefly among them BPM. Here, stakeholders from diverse backgrounds within and between organizations need to grasp the processes and choreographies they are part of, and a wide array of modeling languages and visualizations exists. One of them particularly stands out in the context of smart contracts: the choreography diagram as introduced in the de-facto industry standard modeling framework BPMN (see Sect. 2.3.3). Described as "a type of business contract between two or more organizations" [87, p. 315], they abstract from the private processes of participants and focus on interactions only.

The ostensible connection to legal or smart contracts, however, has since only been casually explored [2]. In this section, we will apply the smart contract metamodel to assess whether BPMN choreography diagrams may be considered adequate visual representations of smart contracts.

Table 3.4: Mapping of smart contract model aspects to their representation in BPMN choreography diagrams

Smart contract metamodel		*Choreography diagram*
Parties	P	Participant bands
Data sources		
– Parameters	D^P	*n/a*
– Variables	D^V	Messages
– External	D^E	*n/a**
Legal relations	L	*n/a*
Actions	A	
– Autonomous		(Intermediate events, gateways)
– Non-autonomous		Choreography tasks
Performance constraints		
– Causal	C^C	Sequence flow, gateway types
– Temporal	C^T	Timer event defs.
– Data	C^D	Conditional flows, conditional event defs.

* May be implicitly qualified within arbitrary formal expressions

3.3.1 Element Mapping

Some components of a smart contract model $\mathcal{M} = (P, D, L, A, C)$ can be mapped to notational elements of BPMN choreography diagrams as shown in Tab. 3.4. As an example, the parties P are a first-class citizen of choreography diagrams represented by participant bands.

Data sources are not as easily mapped, though, due to the fundamental lack of a "mechanism for maintaining any central [choreography] data" in choreographies [87, p. 319]. Thus, choreography diagrams do not have a notion of parameters D^P which can be provided upon instantiation. However, choreography diagrams do support variables D^V and external data sources D^E to some degree: Data from messages can be used in conditions attached to conditional flows going out of exclusive gateways, and may be overwritten and updated by subsequent messages [87, p. 372]. External data sources may be implicitly modeled by referencing them in the formal expression attached to conditional event definitions. The standard gives the example "S&P [stock market index] changes by more than 10% since opening" [87, p. 240], which would refer to some external data source providing the current value of the index.

Actions and constraints are perhaps most elaborately represented in BPMN choreography diagrams, not least due to their inherently behavioral nature. In contrast, legal relations are ostensibly missing entirely, as is the case in the entirety of the BPMN standard for that matter.

3.3.2 Actions and Constraints

We differ between autonomous actions and non-autonomous actions, the latter of which are represented using choreography tasks: A choreography task describes a message exchange from a sender to a receiver, optionally with a second response message afterwards. These message exchanges are deliberately and intentionally initiated by some party, i.e., the sender for the request and the receiver for the response message, constituting an action in the sense of a smart contract.

Fig. 3.6 shows an example, in which the choreography task n_2 labelled "Issue full voucher" may be interpreted as an action similar to a_3 in the train ticket running example $\mathcal{M}_{ticket}$. More properties of the action may also be derived, like the set ChoiceOf of parties responsible for the action—in this case the participant RC specified in the initiating participant band.

Autonomous actions, in turn, encapsulate behavior that the smart contract enforces and is responsible for. In choreography diagrams, this description fits most other flow nodes except for choreography activities, that is, intermediate events and gateways which represent behavior that is independent of any party and should still be enforced. That is, some intermediate events and gateways can be mapped to an autonomous action, like the event n_1 in Fig. 3.6.

Of course, these elements have complex definitions and semantics by themselves. This is where performance constraints come into play: Simple causal constraints C^C between two actions are explicitly modeled using sequence flows. Gateway types may also induce more complex causal constraints, like fork and join behavior in case of the parallel gateway. Temporal constraints C^T are attached to autonomous actions as specified in the timer event definition. Lastly, data constraints C^D are mostly represented in conjunction with causal constraints using conditional flows and their attached expressions. Conditional events with their associated event definition likewise give rise to such constraints.

In our previous work, we have shown that groups of modeling elements in a BPMN choreography diagram—so-called *execution units*—may be bundled and enforced together in one blockchain transaction [70]. This technique of bundling could also be used here to merge subsequent autonomous actions and achieve a more compact smart contract representa-

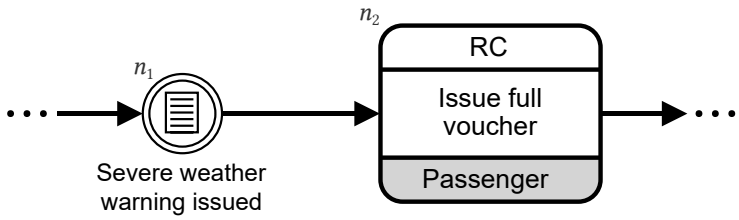


Figure 3.6: Excerpt from a choreography diagram with elements representing types of smart contract actions

tion. Since this is not pertinent to the core research questions of this thesis, though, we will not go into further detail on this.

3.3.3 Legal Interpretation

From a modeling perspective, the facet of legal relations remains largely unexplored in BPMN [87]. Yet, choreography diagrams carry a very concrete legal meaning with them: They encapsulate the steps that the participants have to take in order to achieve a common business goal. The diagram is the normative blueprint for all corresponding interactions, essentially encapsulating the obligations, permissions, and powers of the participants along the way. These legal relations are hidden in the choreography diagram notation, though. To this end, we propose a novel interpretation of choreography diagrams which brings to light these legal relations [65]:

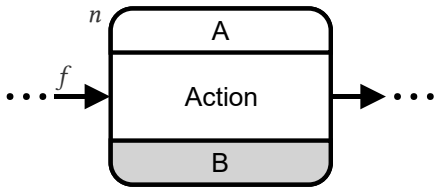
Norms of conduct and liberties, by definition, pertain to concrete behavior of a party (see Sect. 2.1.2). Ultimately, interactions are the only instance of concrete behavior of a party in choreography diagrams, specifying at which points control and data is transferred to fulfill the expectations of each respective participant. The interactions form a set of interfaces, and we consider these interactions to have legal relevance more so than the internal activities within a participant.

We identify three basic patterns of norms of conduct (see Fig. 3.7). An obligation is by far the simplest pattern, and represented by a single choreography task n (see Fig. 3.7a). A participant A is supposed to send a message to a participant B. As such, n represents an obligation of A to perform some "Action" as testified by the message in face of B, i.e., a $\mathsf{Obligation}_{\mathrm{A,B}}(\mathit{Action})$.

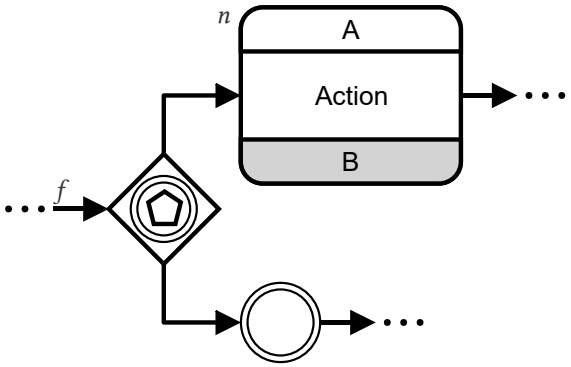
Permissions can be represented by event-based gateways, on the other hand. Figure 3.7b shows an example. This time, the action as represented by the choreography task n is not obligatory, but rather up to A to decide for since there is an alternative path in the model through the intermediate gateway. That is, the choreography will not necessarily get stuck if A does not perform the action, depending on the concrete event definition, representing a permission $\mathsf{Permission}_{\mathrm{A,B}}(\mathit{Action})$.

Lastly, a liberty $\mathsf{Liberty}_{\mathrm{A,B}}(\mathit{Action})$ is similarly represented with two opposed choreography tasks n_1 and n_2 following an event-based gateway, where one is the logical negation of the other. That is, the participant A may both perform the action or explicitly *not* perform the action, i.e., do the opposite. This is a common pattern in such diagrams, for example when accepting or rejecting an order. For all patterns, the correlative rights and no-rights are obtained automatically.

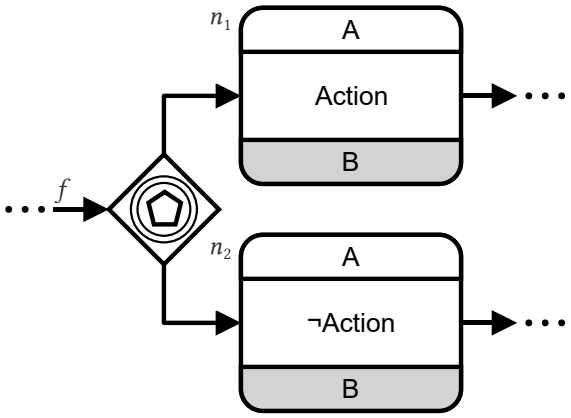
These patterns allow the derivation of a set of legal relations L present in the model. During the execution of the choreography, they also serve to determine the set of currently holding legal relations. When implementing a token-based semantics, a token on any of the incoming sequence



(a) Obligation BPMN pattern



(b) Permission BPMN pattern



(c) Liberty BPMN pattern

Figure 3.7: Patterns of norms of conduct

flows f would lead to the legal relation being considered binding in that choreography state.

Of course, these deduction rules do not cover all legal relations. Indeed, there are legal relations like norms of power which do not have a direct counterpart in choreography diagrams. No-rights and other norms of conduct stating that something must *not* be done similarly are not represented explicitly. Such legal relations would require the introduction of new modeling elements [2]. Still, it is evident that smart contracts and business process choreographies as specified in BPMN exhibit a certain degree of overlap. We use this circumstance in the following chapter when devising a management system for smart contracts closely aligned with BPMSs.

Chapter 4

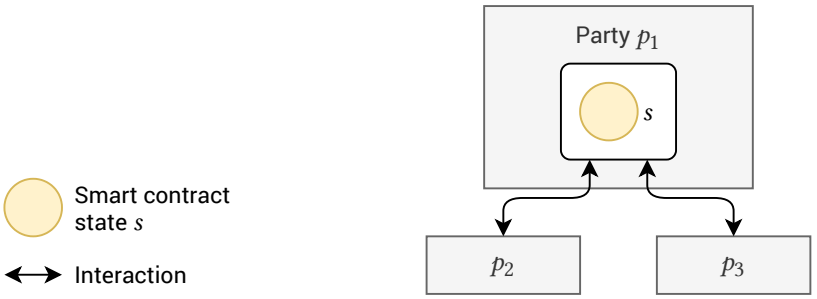
Smart Contract Management Systems

The smart contract metamodel and the attached operational semantics allow the reasoning about the structure and behavior of smart contracts. By doing so, they also pose concrete functional requirements as to what a dedicated management system for smart contract modeling and enforcement must be capable of. Based on these insights, we introduce an architecture based on blockchain technology for what we call a Smart Contract Management System (SCMS). The task of an SCMS is to facilitate all phases of the contract lifecycle as applied to smart contracts; from their initial modeling and negotiation, to storage and performance, up until supporting dispute resolution.

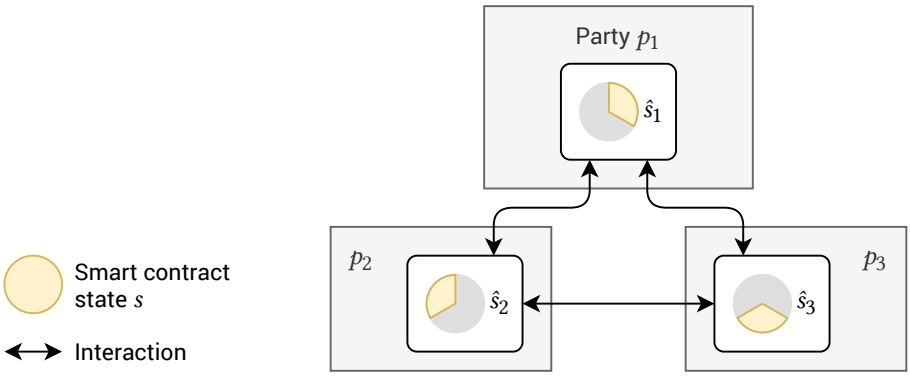
The similarity in name and scope to the notion of Business Process Management Systems (BPMSs) is no coincidence. We draw from the rich set of techniques and solutions already established in the BPM domain as a recurring theme throughout this thesis. In this case, we use the fundamental architecture of BPMSs (see Sect. 2.3.2) as a basis for the SCMS approach, and adapt or extend it where appropriate.

To this end, we will first gradually develop the general enforcement approach the SCMS relies on, namely enforcement via a set of multiple heterogeneous blockchain networks running DApps (see Sect. 4.1). We will then propose a non-exhaustive list of functional requirements for an SCMS based on this enforcement approach and the contract lifecycle (see Sect. 4.2). Lastly, we will introduce and describe a component-based system architecture for an SCMS (see Sect. 4.3).

Parts of this chapter are based on our previous work, in which we (i) discussed enforceability considerations of business process choreographies using blockchain technology [70], and (ii) introduced a novel architecture for multi-chain BPMSs geared towards choreographies [71].



(a) Centralized approach



(b) Distributed approach

Figure 4.1: Smart contract enforcement approaches adapted from BPM orchestration and choreography enactment [112]

4.1 Blockchain-Based Enforcement

According to its definition (see Sect. 2.1), a smart contract is enforced in two ways: (i) partly via legal enforcement of its terms by law, and (ii) partly through automated tamper-proof execution of computer code. The SCMS, being a software system, will primarily drive the latter mode of enforcement.

4.1.1 Non-Blockchain Baseline

Before we introduce the use of blockchain technology for smart contract enforcement, we want to highlight how traditional electronic contracts are managed and enforced using non-tamper-proof means. One of these, the centralized approach, was already discussed in the introduction to this thesis: When buying a digital train ticket with the German national railway company, they store all information related to the ticket on their end. The passenger may receive a digital snapshot, but the single source of truth is maintained by the railway company. To interact with the contract, other parties have to use the interfaces provided by the railway company.

Fig. 4.1a shows this enforcement scheme for generic smart contracts and three parties $P = \{p_1, p_2, p_3\}$. The smart contract state s of some smart contract instance S is stored within p_1, e.g., in an enterprise resource plan-

ning system. This approach is very similar to the orchestration approach of executing business processes with BPMSs in BPM, where one party may be chosen as the designated "orchestrator" of an inter-organizational choreography.

Another method to enforce a smart contract is the distributed approach (see Fig. 4.1b), similar to pure choreographies in BPM. In this case, there is no central orchestrator. Instead, each party manages a part of the smart contract and its state—, and interactions are used to synchronize. This results in a distribution of the state of $\mathcal{S}$: p_1 owns and manages $\hat{s}_1$, p_2 owns $\hat{s}_2$, and p_3 owns $\hat{s}_3$. Combined into one, the state fragments yield s. The exact distribution of the state must be somehow specified, e.g., in the model. No party has power over the whole smart contract, but no party has a view of the whole state either [70].

In the train ticket scenario, choreography was the common way of enforcing contracts of carriage when paper tickets were still common. Passengers would buy a ticket in interaction with the railway company. During the journey, a conductor would check the information printed on the ticket, and validate it. There is no communication with the railway company anymore, and the whole state of the contract is generally unknown to them, e.g., whether the passenger actually ended up taking the train.

Both approaches have a major downside, though, which makes them unsuitable for smart contract enforcement. There is no guarantee that the smart contract state s or operational logic running locally is not tampered with—either the orchestrator $p \in P$ needs to be trusted with the whole state s, or all of the parties P with their state fragments, respectively. This is summarized in Tab. 4.1. Blockchain technology and DApps provide a solution to this dilemma by ensuring a completely independent, tamper-proof execution and storage environment.

4.1.2 Single-Chain Approach

Given a smart contract model $\mathcal{M} = (P, D, L, A, C)$ and an instance $\mathcal{S}$, the general idea of blockchain-based enforcement is to store the entire state $s \in \mathbb{S}$ of $\mathcal{S}$ within a DApp Ð on a blockchain network $\mathcal{N}$. This essentially makes Ð the sole orchestrator of the smart contract [70]. Ð becomes the single source of truth, logically centralizing the smart contract $\mathcal{S}$ within a distributed, trustless environment. The state s benefits from the fun-

Table 4.1: Key differences between smart contract enforcement strategies

Approach	*Storage type*	*Storage location*	*Tamper-proof*
Centralized	Whole	Participant $p \in P$	No (p)
Distributed	Fragmented	Participants P	No (all of P)
SCMS single-chain	Whole	DApp Ð	Yes
SCMS multi-chain	Fragmented	DApps $Ð_1$, ..., $Ð_m$	Yes

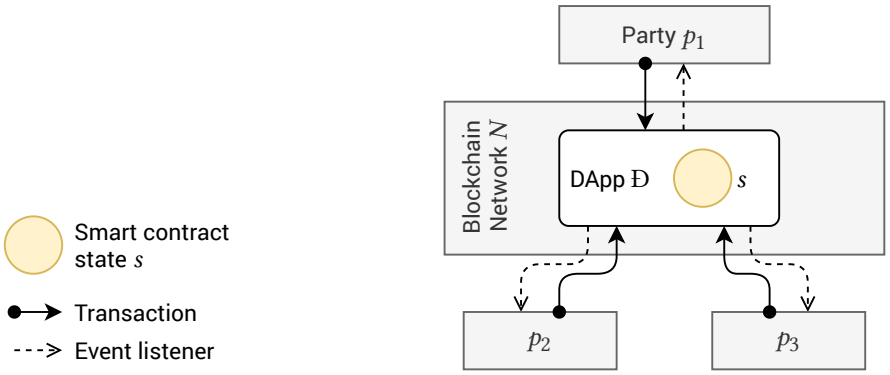


Figure 4.2: Smart contract enforcement approach using a single blockchain network

damental integrity guarantees of the blockchain network, meaning each party can independently ensure its validity [115].

A scheme of this approach is shown in Fig. 4.2. The interaction between the parties is funneled entirely through the DApp Ð on the blockchain network $\mathcal{N}$. Every interaction concerning the smart contract, i.e., the performance of actions, is contained in a transaction tx targeting Ð. The immutability property of Ð (see Sect. 2.2.4) guarantees a tamper-proof execution. The performance of each action is automatically logged in the ledger, guaranteeing non-repudiation.

Zooming in, each action $a \in A$ is wrapped in a blockchain transaction $tx(a)$ (see Fig. 4.3). For example, consider a state transition

$$s \xrightarrow{o/a} s'$$

of S with an operating environment state $o \in \mathbb{O}$. One or all of the parties $p \in \mathsf{ChoiceOf}(a)$ would be responsible for creating $tx(a)$, and submitting it to the blockchain network $\mathcal{N}$. This procedure is shown in Fig. 4.3. The state s stored in Ð would, of course, not change immediately. Instead, the transaction must first be picked up by a miner, and be included in a block

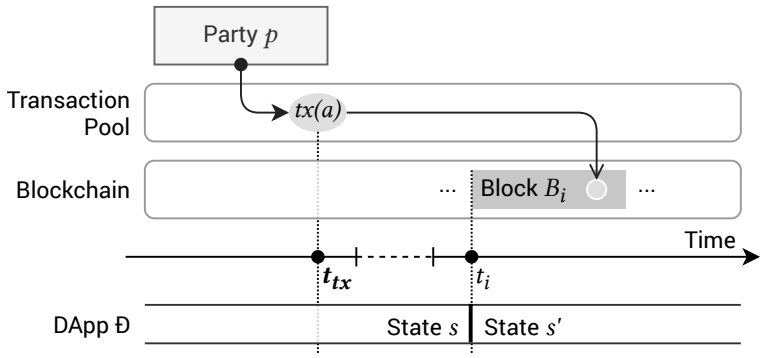


Figure 4.3: Mapping of the performance of an action of a smart contract to a blockchain transaction

B_i. Ð checks whether a is enabled, and if so, executes all the code associated with a arriving at state s'.

Transaction-Driven Semantics

Moving the smart contract state and logic to a DApp has significant implications for the performance of smart contracts. If Ð is to enforce $\mathcal{S}$, it needs to be able to correctly implement the smart contract semantics. However, two limitations of smart contracts, as stated in Sect. 2.2.4, pose substantial challenges:

- Due to the **non-continuity property**, Ð is only active within transactions during the mining of a new block, and otherwise lies dormant. State transitions of the smart contract must, therefore, (i) wait until they are included by a miner, causing a delay (see Fig. 4.3), and (ii) be explicitly triggered by an outside system or party.
- Due to the **isolation property**, Ð can not access any systems or data outside the blockchain. Thus, any access to the valuation ϕ external data sources D^E is severely hampered. In our architecture, we will make use of oracle patterns to somewhat circumvent this limitation.

The impact of these limitations, especially on the performance of autonomous actions as well as on detecting external events, will be the main topic of Chapter 6. For now, we assume that any delays are minimal and within the tolerance of all parties, and that autonomous actions can be supported.

Confidentiality Issues

This leaves a more immediate issue to deal with concerning the transparency of blockchains and the resulting visibility of $\mathcal{S}$ and its state s within Ð: Assuming a public blockchain network is used, the smart contract, its state, and all associated communication is not only visible to all parties, but also to all other members of the blockchain network $\mathcal{N}$ and the general public. Arguably, this high level of transparency is one of the most fundamental features of blockchain technology, and automatically enables full auditing, monitoring, and non-repudiation capabilities. These may especially be helpful when it comes to a legal enforcement of the smart contract in front of a court.

However, for many smart contracts it may be an obstacle: The underlying operational logic, the contract data and all messages, as well as the identity of the parties are not always supposed to be public or even shared between all immediate parties to the contract. Yet, the DApp needs access to, e.g., the values of the internal data sources D^P and D^V to perform actions autonomously. Using permissioned blockchain networks may be

considered to mitigate this to some degree, but will result in a lack of flexibility. As such, a SCMS must strike a balance between ensuring the independence and power of the DApp while ensuring privacy to varying extents, for which we propose a multi-chain approach.

4.1.3 Multi-Chain Approach

In practice, it becomes increasingly evident that different domains call for very individual requirements when it comes to blockchain networks [108]. As a result, initiatives with various goals have been formed: Estonia, for example, makes use of a custom blockchain network powered by the KSI blockchain[1] in some of their government agencies. Australia, likewise, aims at establishing a national blockchain network[2] for various uses, such as aiding businesses in achieving regulatory compliance as well as credentialing in the education sector. In industry, especially the supply chain sector is exploring the use of blockchain technology for securing information and provenance across supply chains [64], one such productive example being TradeLens[3].

This emerging multi-chain environment poses challenges for smart contracts, but also inspires a smart contract enforcement approach: When using multiple blockchain networks to enforce the same smart contract, each part of the smart contract can be configured separately. Sensitive parts of the smart contract may be enforced on highly restricted consortium blockchains, while less sensitive parts may be moved to public blockchains. Other properties of the blockchain network may also be exploited and configured, e.g., when optimizing the cost of storage and transactions throughout the smart contract's lifecycle.

Figure 4.4 shows an example of how such a multi-chain environment may look like for the train ticket scenario. First, train tickets are subject to local laws, for example regarding customer protection. A national blockchain network $\mathcal{N}_1$ may be used to track additional data and complaints about the passenger's experience to handle compensation and refund schemes. Railway companies often cooperate with each other and merge their infrastructure to offer better deals and more attractive routes. Such strategic partnerships form consortia: The railway companies may, as a consequence, decide to run their own railway consortium blockchain network $\mathcal{N}_2$ to keep track of tickets and funds. The same general principle applies to payment providers, who may run a payment consortium blockchain network $\mathcal{N}_3$.

The passenger needs access to all three blockchain networks, while the railway company only accesses $\mathcal{N}_1$ and $\mathcal{N}_2$ and the payment provider only $\mathcal{N}_3$. As a consequence, the payment information of the passenger can

[1] https://e-estonia.com/

[2] https://www.industry.gov.au/data-and-publications/national-blockchain-roadmap

[3] https://www.tradelens.com/

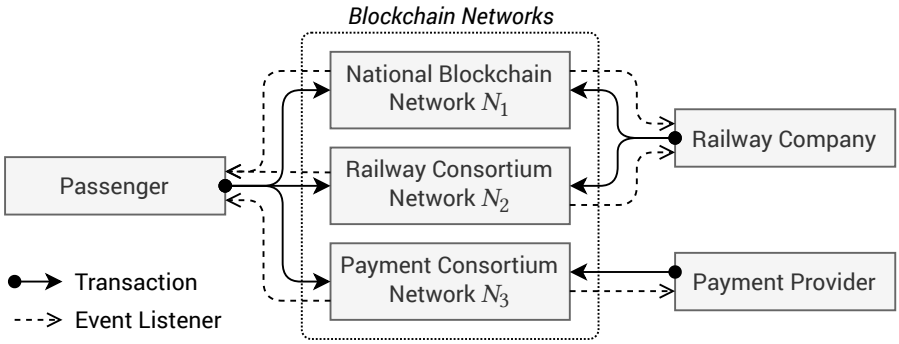


Figure 4.4: Blockchain networks potentially involved in the train ticket scenario

be hidden from the railway company, and the ticketing details can in turn be hidden from the payment provider. This allows the creation of strict need-to-know guidelines across a smart contract, balancing the tradeoff between transparency and trust and the need for privacy and confidentiality.

The consequences for the enforcement of the smart contract $\mathcal{S}$ are significant. Like in the choreography enforcement approach, the state may again be fragmented and distributed, this time across a number of blockchain networks $\mathcal{N}_1$, ..., $\mathcal{N}_m$ and corresponding DApps $Đ_1$, ..., $Đ_m$. Figure 4.5 shows this general scheme, and how the state fragments $\hat{s}_1$, ..., $\hat{s}_m$ are maintained by one DApp each. Each participant needs to access and manage potentially multiple blockchain networks.

A major challenge in this area is that of *blockchain interoperability*, i.e., the degree to which heterogeneous and homogeneous blockchain networks can exchange information, synchronize their states, or trigger actions within each other [57]. Such capabilities are vital when working to-

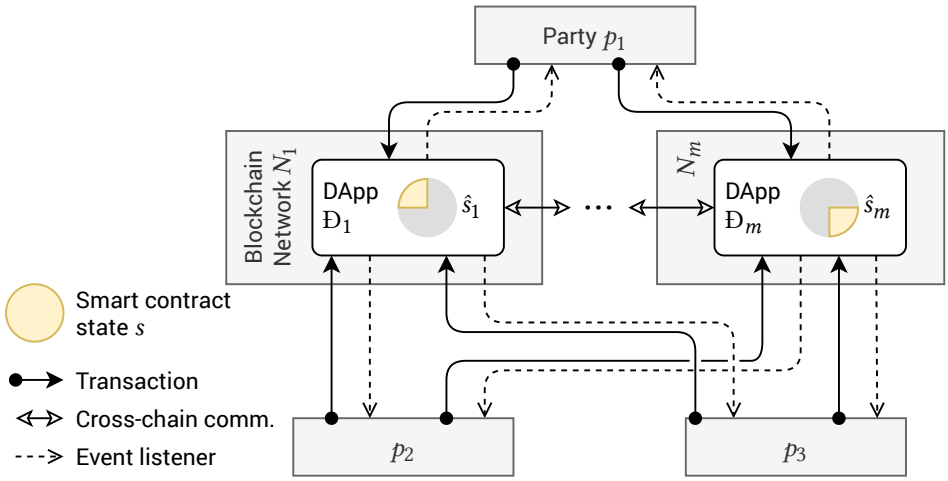


Figure 4.5: Smart contract enforcement approach using multiple blockchain networks

wards a multi-chain smart contract enforcement approach to ensure consistency between the smart contract state fragments.

In a recent survey exploring the state of the art in blockchain interoperability, Belchior et al. found a wide array of approaches which vary in their complexity and applicability [14]. In the context of smart contracts, it is critical whether additional trust is needed to allow for blockchain interoperability. Many *sidechain* approaches, for instance, in which assets on one blockchain are locked and free corresponding "pegged" assets on another blockchain, work using third-party validators. The same may be true for *blockchain relays*, which often rely on a trusted party managing the interactions. Second layer consensus mechanisms and blockchains deliberately built to support interoperability may remove this restriction, though, and allow for interoperability between different blockchain networks without sacrificing the inherent guarantees of using a single network [14]. In the remainder of this chapter, we will take on a more agnostic view on blockchain interoperability, and aim not to prescribe any single approach.

4.2 Functional Requirements

Since smart contracts are a relatively novel concept it is difficult to determine a set of functional requirements for a SCMS that covers all practical needs. However, in BPM the situation is different: BPMSs have been in development and productive use for decades [31, 112]. As such, there is a rather clear picture of the features expected from a BPMS [91] (see Sect. 2.3.2), which we aim to transfer to the case of smart contracts.

Figure 4.6 lends credibility to this approach: A BPMS facilitates processes and choreographies through their entire lifecycle. Fortunately, the business process lifecycle (see Sect. 2.3) aligns well with the contract lifecycle (see Sect. 2.1). Essentially, the contract is designed, analyzed, configured, enacted, and evaluated in a similar way to a process. Only the final dispute resolution and termination phases of the contract lifecycle are not explicitly contained in the business process lifecycle. Dispute resolution is usually not necessary when processes within a single organization are concerned—at least not on a legal level including courts or arbitrators. The absence of the termination phase is due to the differing perspective between the two lifecycles: Whereas the business process lifecycle describes the ongoing management of a repeated process in an organization, the contract lifecycle describes the phases of an individual, atomic contract.

Still, our assumption is that the features of a BPMS regarding process and choreography models can be analogously moved to an SCMS regarding smart contract models. Where applicable, we will add new features to fit the requirements of smart contracts.

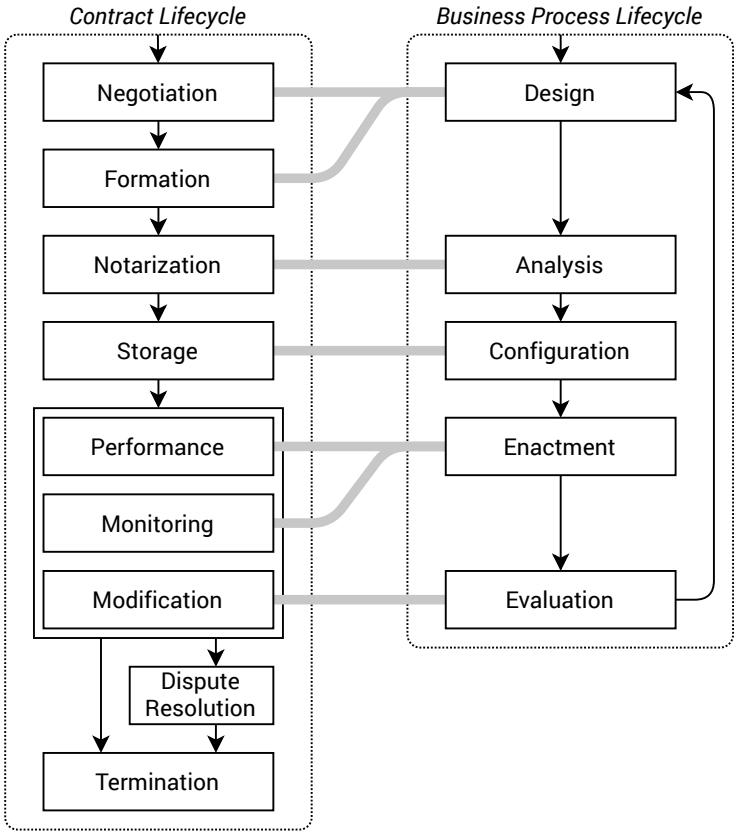


Figure 4.6: Approximate relationship between the contract lifecycle by Governatori et al. [39] and the business process lifecycle by Weske [112]

4.2.1 Negotiation and Formation

The negotiation and formation phases of the contract lifecycle roughly correspond to the design phase of the business process lifecycle. This includes the identification and modeling of a business process, and is primarily covered by the process modeling tool of the BPMS. Adopting this functionality for the SCMS yields the following functional requirements:

F_1 – *Create and modify smart contract models.*

Of course, the core functionality of initially creating and modifying existing smart contract models must be given. The smart contract models should be complete in that they contain all relevant information according to our metamodel (see Sect. 3.1), but could be drafted in any notation or formalism.

F_2 – *Store smart contract models.*

The SCMS should be able to store smart contract models in a smart contract metadata storage. The repository should be protected with security and authentication measures so that users can only access data they are allowed to see.

F_3 – *Share smart contract models.*

Creating a smart contract model is a collaborative procedure. Safely sharing the smart contract models between all parties for negotiation purposes is essential.

We do not prescribe any specific technology or procedure for creating and sharing the smart contract models. In research, there are even approaches which employ blockchain technology at this stage, for example, to build the entire model on the blockchain or to find consensus on which parts of a model need to be adapted during negotiation [49]. Naturally, no step should compromise the integrity of any smart contract model.

4.2.2 Notarization and Storage

The notarization and storage phases of the contract lifecycle overlap with the analysis and configuration phases of the business process lifecycle. A deployment in the sense of a business process most directly corresponds to the signing of the contract, since before there is no legally binding agreement yet. At the time of signing, the configuration—so to say—of the contract has to have finished already as well. This is reflected in the feature requirements of the SCMS:

F_4 – *Analyze smart contract models.*

Analysis tools should be available to make sure the smart contract model is free of issues and mistakes. This could pertain to structural issues, such as infinite loops of immediately enabled autonomous actions. Targeted analyses regarding the actual content of the smart contract and its terms are also conceivable, e.g., checking the smart contract actually makes good on all offers initially negotiated.

F_5 – *Configure privacy and security parameters.*

Since smart contracts are subject to critical privacy and security considerations, these should be configurable on a per-model basis. This could for example mean to specific configuration of a consortium blockchain network, or the use of multiple blockchain networks with different properties for different parts of the smart contract.

F_6 – *Sign smart contract models.*

Once the parties have agreed on a smart contract model, they need to sign it to demonstrate clear intent to enter the encoded agreement. This may be facilitated using cryptographic means like electronic signatures. This step could also be included in the act of deploying corresponding DApps signed by the accounts of all parties.

F_7 – *Instantiate smart contracts.*

After the configuration and signing is done, a smart contract instance may be created from a model. This includes the deployment of all required DApps on potentially multiple blockchains and properly interlinking them. At this point, code needs to be generated from the smart contract model, or suitable interpretation frameworks must be set up.

F_8 – *Share access to smart contracts.*

Once deployed, parties need to have access to the smart contract. The SCMS should facilitate the exchange of correlation information regarding instances, e.g., by sharing addresses to all DApps.

Note that it is not the goal of this thesis to solve all issues of smart contracts on a deployment level. For example, we do not provide any methodology, neither automated nor manual, to distribute a smart contract model among multiple blockchain networks. In previous work, though, we have shown that the BPMN choreography diagram notation can be extended with visibility constraints to aid privacy at design-time [70], which could be used as an indication for a distribution algorithm. Other design-time approaches have also been proposed, such as the privity spheres introduced by Köpke et al. [63].

Regarding the implementation of pre-configured privacy parameters, there is a lack of support in practice for implementing most efforts towards completely or partially private DApps without sacrificing important functionality. For example, a DApp on a public blockchain network can not perform any calculations based on encrypted data, since it would require the decryption keys—which would automatically become public. Approaches like homomorphic encryption or Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zkSNARK) [15, 32], which could empower DApps to perform calculations without disclosing any information, are not yet ready for widespread use. The goal is rather to keep privacy in mind during the overall abstract design of a SCMS, so that future developments can be included.

4.2.3 Performance and Monitoring

Once the smart contract is signed and stored—that is, instantiated—, the actual performance in which the parties fulfill their respective obligations begins in the contract lifecycle. This performance phase, along with the parallel monitoring phase, are captured by the enactment phase of the business process lifecycle. In a BPMS, the execution engine is the main driver of this phase, but monitoring tools are needed as well. Together, the following functional requirements emerge:

F_9 – *Implement smart contract semantics.*

The code generated for the DApps or the interpretation mechanisms contained in the DApps need to make sure that the correct semantics of smart contracts are enforced. An action may only be performed if it is enabled, and autonomous actions must be performed immediately after they become enabled. Performance constraints must be monitored and evaluated to ascertain this.

F_{10} – *Interact with smart contracts.*

Actions which are enabled by the smart contract need to be visible to the parties, and interfaces for human actors must exist to perform these actions. In particular, this means that transactions need to be created and sent to the appropriate blockchain network to trigger any associated code in the DApps.

F_{11} – *Monitor smart contracts.*

Since smart contracts may be distributed among many blockchain networks and receive interactions from other parties continuously, the SCMS must provide appropriate monitoring information. This includes the current smart contract state, potentially redacted according to the parties' access authorizations.

F_{12} – *Connect to external oracle services.*

The SCMS must facilitate and manage interactions with external oracle services. This interaction is not necessarily direct, but may also be the result of generating specific code during the code generation phase.

F_{13} – *Connect to blockchain interoperability approaches.*

The SCMS should be designed to enable various blockchain interoperability approaches. This includes provisions in the code generators to allow for the use of blockchain relays, or specific adapters accessing overlay blockchain networks [14].

As discussed in the previous chapter, physically forcing parties to perform a non-autonomous action they are obliged to is not possible. Thus, enforcement via tamper-proof execution of code is principally limited in its scope. Still, DApps can be used to ensure that a party does not perform an action which is not enabled, and that autonomous actions are performed as soon as possible.

4.2.4 Modification, Disputes, and Termination

Lastly, contracts may either call for modification, dispute resolution, or termination during their regular runtime. Whereas modification aligns with the evaluation phase of the business process lifecycle, dispute resolution

and termination have no direct counterpart. The three phases of the contract lifecycle have in common, though, that they concern the relationship of the parties, and a consensus about how to proceed with a running smart contract outside its original terms. We identify the following functional requirements:

F_{14} – *Modify or terminate smart contracts.*

Contract terms may be altered during the runtime of the contract, if all parties agree. An SCMS should support this to some degree. Practical implementations using blockchain technology are limited by the immutability property of blockchain data, including the DApp code. Patterns exist to circumvent this limitation, for instance using registries and proxy references which redirect function calls to newly deployed smart contracts [61].

Smart contracts may also be amicably terminated before the specification calls for it, which is a special case of modification. The execution layer needs to make this possible, and avoid any autonomous actions from being performed after the termination [39].

F_{15} – *Extract auditing information from smart contracts.*

If parties in a contract feel the need to dispute any part of the contract, this is usually done in arbitration or litigation proceedings. The SCMS must provide authenticated logs and auditing information, which aids the legal enforcement of the smart contract. This auditing information can also be used for evaluation and later process mining activities [62].

Note that we consider blockchain migration, that is, moving one or all DApps to different blockchain networks without influencing their operation, to be a modification of the smart contract. Bandara et al. propose a number of patterns to this end, which adapters may implement [13].

These requirements are not meant to be exhaustive, but provide for a first design of an SCMS that is extensible for future developments.

4.3 System Architecture

Based on the multi-chain enforcement approach and the functional requirements, we devised a blueprint system architecture for generic SCMS implementations [71]. The architecture is inspired by the BPMS architecture introduced in Sect. 2.3.2.

Note that the term SCMS refers not to a single component, but essentially the entirety of the distributed components running on-premise at the parties or on the blockchain network. The overall system architecture from the perspective of a single party is shown in Fig. 4.7. In the following, we will describe these components separately.

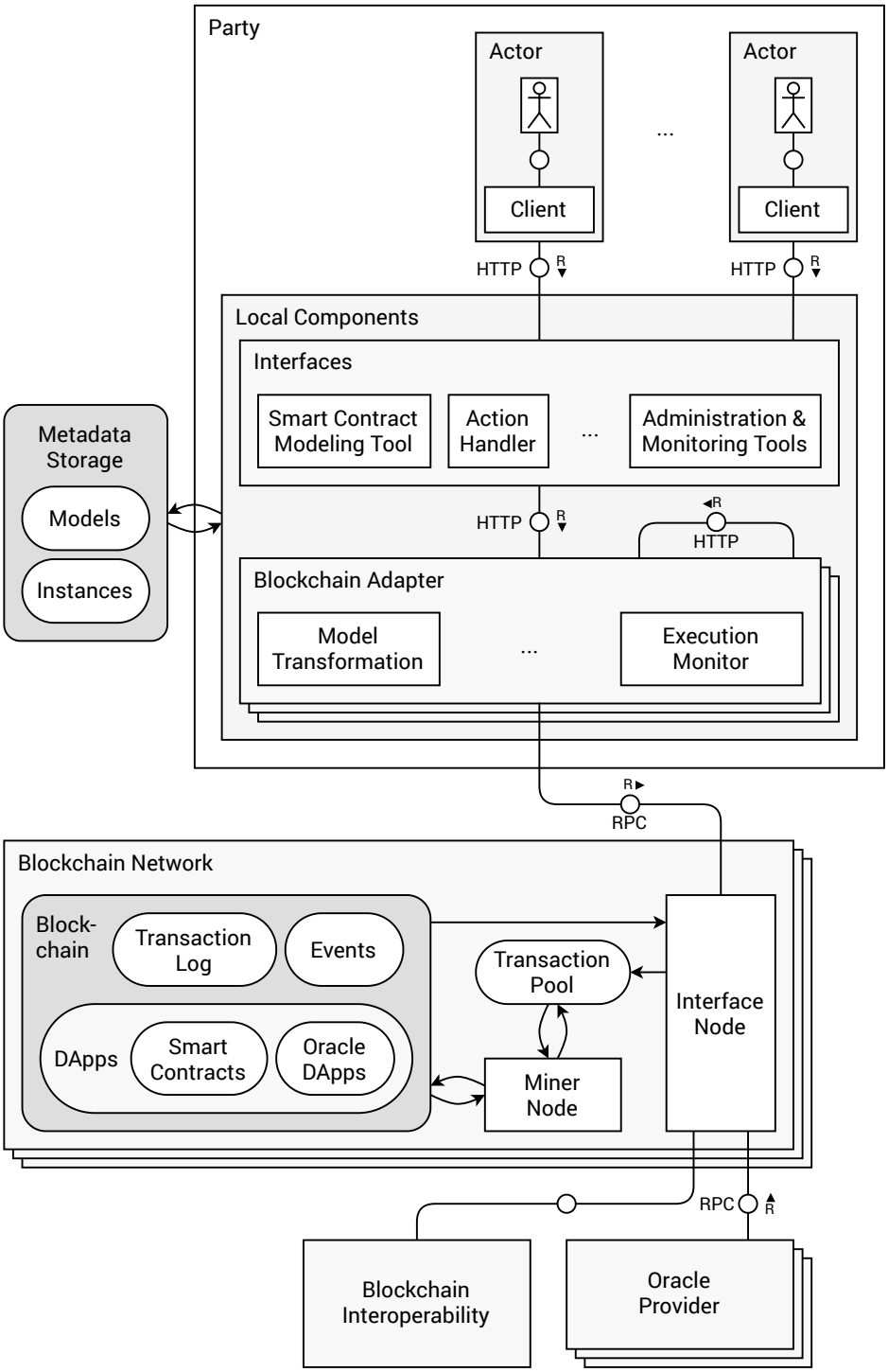


Figure 4.7: Reference architecture of a generic SCMS approach

4.3.1 Local Components

All local components are deployed on-premise at each party depending on their involvement in the smart contract.

Interfaces

The interface components provide the frontend of the SCMS, and are used by the human actors at each party to create and interact with smart contracts in some form, spanning all phases of the contract lifecycle. In the architecture diagram, three examples are shown:

A *smart contract modeling tool* is used to create smart contract models in whichever concrete notation is chosen in a particular implementation (F_1). For example, in our prototypical implementation Mantichor (see Chapter 5), BPMN choreography models visualized using BPMN choreography diagrams were used. The modeling tool may, of course, be arbitrarily complex and include analysis features (F_4) or collaborative modeling features among the contract parties (F_3) [49]. Further, if privacy and security parameters are included in the smart contract model, these may be configured as well (F_5).

The *action handler* is the main gateway to the smart contract's execution (F_{10}). As per the semantics (see Sect. 3.2), the current state of the smart contract enables a number of actions which are the choice of a party. The action handler allows these parties to view actions enabled for them, and properly perform them. This is done via a blockchain transaction through a blockchain adapter, supplying the necessary information to the DApp for tamper-proof processing.

Lastly, *administration and monitoring tools* are used to keep track of running smart contracts, since parties could be involved in more than one at the same time. A list of active smart contracts, an overview of available actions in each of them, statistics about the smart contracts, and means of modifying or extracting information from smart contracts (F_{14}, F_{15}) are all offered via this component.

Blockchain Adapters

All of the interface components need access to the DApps actually enforcing the operational logic of the smart contract. To manage the complexity of handling many blockchain networks with potentially varying interfaces, we add another layer between the interfaces and the DApps following the adapter pattern. The idea is to provide a custom adapter component for each distinct blockchain network which knows how to generate code or create transactions for that particular blockchain network.

Blockchain adapters are used together with dedicated interface components like the smart contract modeling tool to allow the signing and instantiation of new smart contract instances via the deployment of DApps (F_6, F_7). The challenge here is to find a common interface which unifies

the blockchain networks while also providing all the necessary information to the frontend components. We will talk about a concrete proposal for this interface in the course of introducing Mantichor (see Chapter 5). In general, though, we see the need for two components at least:

A *model transformation* takes a smart contract model as an input and generates some kind of DApp specification, usually code in a domain-specific programming language (model to text transformation). The model transformation may be enhanced with analysis and verification tooling to ensure a semantics-preserving translation of the model semantics to the DApp (F_4). One of two approaches is generally used:

- A *code generator* transforms the smart contract model into a DApp specification which can then be deployed to the blockchain network. The specification may be in any format that the target blockchain network accepts, e.g., Solidity scripts for Ethereum [114] or Kotlin applications for Corda [45].
- A separate *interpretation infrastructure* is available on the blockchain network, which requires the smart contract model to be converted to an intermediate format the interpreter understands.

In this step, adapters may also require some coordination between each other to enable later interoperability between DApps. For example, specific event types for cross-chain communication may be required to be generated, which the adapters share between each other [14]. We account for this possibility by allowing adapters to access other adapters using their regular APIs. Additionally, a generic *blockchain interoperability* component symbolizes the varying capabilities of blockchain networks to interact with each other (F_{13}).

Second, an *execution monitor* is used to observe any activity of the DApp (F_{11}). For example, blockchain networks like Ethereum offer an event layer, in which DApps can emit events which outside entities may pick up following a listener pattern. The execution monitor may serve as such a listener, and correlate events with smart contract instances to display in the frontend components. Further, the execution monitor may also provide the state of the DApp to the interfaces for analysis.

Depending on the implementation of the DApps, an execution monitor may also be tasked with polling for the enablement of autonomous actions, and triggering appropriate transactions for them as soon as possible. We will discuss different strategies of implementing such behavior in oracle providers in Chapter 6.

4.3.2 Metadata Storage

The *metadata storage* stores the associated models and instance information like DApp addresses (F_2, F_3, F_8). The metadata storage is the main means of sharing information between parties *before* the smart contract

is notarized and stored on one or more blockchain networks, or *before* all parties are aware of the required information needed to access the DApps in the first place. Thus, it facilitates the negotiation and formation phase of the contract lifecycle.

Of course, every interaction between the parties which is not pushed through a blockchain network via a transaction poses a potential risk of tampering and manipulation. Concrete implementations must therefore use adequate mechanisms to secure the metadata storage and its interfaces. The degree to which this is necessary depends on the parties involved. If there is a complete lack of trust, even the metadata storage and the eventual negotiation of the smart contract model may be secured via a blockchain network [101].

In this context, note that the SCMS architecture does not arrange for additional off-chain storage of any kind. A common pattern, for instance, is to attach data to DApps by only storing a hash of the data in a DApp, which allows users to validate their local copy [116]. Patterns like this are needed due to the impracticability of storing large amounts of data on many contemporary blockchain networks like Ethereum, since they rely on fully replicating nodes [114]. As a result, they introduce steep fees for data storage which may quickly become prohibitively expensive (see also Sect. 6.5). Research has since been focused on patterns and methods to ensure a secure link between off-chain and on-chain data [46]. Projects like Ethereum Swarm[4] even propose incentive-based decentralized file storage which is enforced through DApps itself.

Using these approaches for smart contracts does not appear adequate, though. The idea of smart contracts is full enforceability through law or tamper-proof code. Introducing data which may be tampered with or even disappear after a while if incentives are not met anymore fundamentally contradicts this idea. In short, we believe that everything that pertains to a smart contract and its enforcement after it has been stored and deployed—that is, its full state and even further attachments—should be contained in a DApp on a blockchain network.

Still, scalability issues exist in practice, and blockchain networks are being developed which contain more appropriate, native means to deal with large amounts of data. A promising approach are zkSNARKs, which allow reasoning about off-chain data from within DApps in a tamper-proof way using complex cryptographic schemes [32]. They are, however, very computationally intensive and may not become viable in the near future.

4.3.3 Smart Contract DApps

The smart contracts are implemented using DApps deployed to one or more blockchain networks (see Sect. 4.1) and contain the actual operational logic adhering to their operational semantics (F_9). Blockchain adapters provide a common API for the interface components to use. To this end,

[4] `https://swarm.ethereum.org/`

each party runs and maintains a local blockchain adapter for each blockchain network they need to interact with. If a smart contract is deployed to multiple blockchain networks, a party thus only needs those adapters required to connect to the blockchain networks they are involved in.

Transactions are sent through an interface node. Individual blockchain adapters may either operate this node themselves, or connect to external services providing an access gateway. An example for such a gateway is Infura[5], which allows users to interact with public Ethereum blockchain networks using a web-based REST API. The adapters also provide monitoring information to any interested parties. That is, they access the blockchain networks event layer and listen for changes, and may also provide other data analysis and aggregation features.

The SCMS makes use of several blockchain technology specific types of patterns [116]. One of these are oracle patterns (F_{12}), which allow smart contracts to access external data sources D^E. Oracles are an important pattern in blockchain networks and are thus also first-class citizens in the proposed SCMS architecture. There are two parts to each oracle, one being the *oracle provider* and one being the oracle DApp. An oracle is often only serving one particular blockchain network, which needs to be taken into consideration when choosing a blockchain configuration for deployment purposes.

[5] `https://infura.io/`

Chapter 5

Proof-of-Concept Implementation

To evaluate key aspects and characteristics of the SCMS approach, we developed a proof-of-concept implementation dubbed "Mantichor". The main goal of Mantichor was to identify whether state-of-the-art blockchain technology is suitable for smart contract management and enforcement, and whether the structure of the SCMS system architecture provides sufficient guidance.

Mantichor was implemented with the help of several Master's students at Hasso Plattner Institute, University of Potsdam, during a dedicated seminar in the Summer Semester 2019, namely: Christian Friedow and Oliver Adameck for the frontend components; Lisa Ihde and Jonas Bounama for the Corda blockchain adapter; as well as Simon Siegert, Tom Lichtenstein and Finn Klessascheck for the Tezos blockchain adapter. Christian Friedow further proceeded to base his Master's thesis on Mantichor.

In this chapter, we first introduce the design of the Mantichor system (see Sect. 5.1) and then go into detail on the blockchain network adapters (see Sect. 5.2). We evaluate Mantichor and discuss its maturity in Sect. 5.3. On a side note, we will also introduce chor-js, a web-based BPMN choreography diagram editor used in Mantichor and independently developed with Anton von Weltzien as part of this thesis (see Sect. 5.4).

Parts of this chapter are based on our previous work, in which we (i) discuss Mantichor in the context of blockchain-based choreography enforcement [71], and (ii) present the core features of chor-js [69].

5.1 System Design

To keep the scope of Mantichor in check, the major design decision to use choreography models visualized using BPMN choreography diagrams as the underlying modeling formalism for smart contracts was taken. This decision had multiple reasons:

- BPMN is an established and well-understood modeling standard. We have further established (see Sect. 3.3) that many aspects of smart contracts are in some way represented in BPMN, and others like actions and legal relations may be derived to some degree.
- A large set of tooling already exists for BPMN, including ready-to-use metamodel and model data structures supported by state-of-the-art technologies. In addition, we were in parallel developing a dedicated web-based modeling tool for choreography diagrams, chor-js (see Sect. 5.4), which could immediately be used in Mantichor.

We therefore believe that using choreography models does not substantially deviate from the general sentiment of the SCMS approach, and that Mantichor is a valid evaluation of its core ideas. As such, Mantichor provides some first steps towards practical and fully-featured SCMS implementations employing blockchain technology.

5.1.1 System Overview

Figure 5.1 shows an overview of Mantichor's overall system design. Components highlighted with a shaded background were originally developed for Mantichor. Their code is available publicly on GitHub[1].

Each party runs their own instance of a dedicated frontend which provides access to the other components via a web browser. The frontend communicates with a share server, which is used to exchange models and instance metadata information between different participants. For Mantichor, we chose to provide access to two blockchain networks: Tezos [38] and Corda [45], resulting in two distinct blockchain adapters and DApp architectures.

A major development goal of Mantichor was to achieve plug-and-play connectivity between the different components without the use of complicated setup procedures. Thus, all components support Docker containers, meaning they can be started and instantiated separate from each other on any supported target platform.

5.1.2 Frontend Components

The frontend of Mantichor allows parties to interact with the system using their web browser, and encapsulates the local interface components of the SCMS approach, namely a modeling tool, action handler, and monitoring tool in a unified application. The frontend is implemented using the Vue.js framework and TypeScript, a strongly typed programming language compiling to ES6-compliant JavaScript usable across many browsers. All choreography models are created and visualized using chor-js.

[1] `https://github.com/bptlab`, projects `mantichor-frontend`, `mantichor-share`, `mantichor-tezos`, `mantichor-corda`

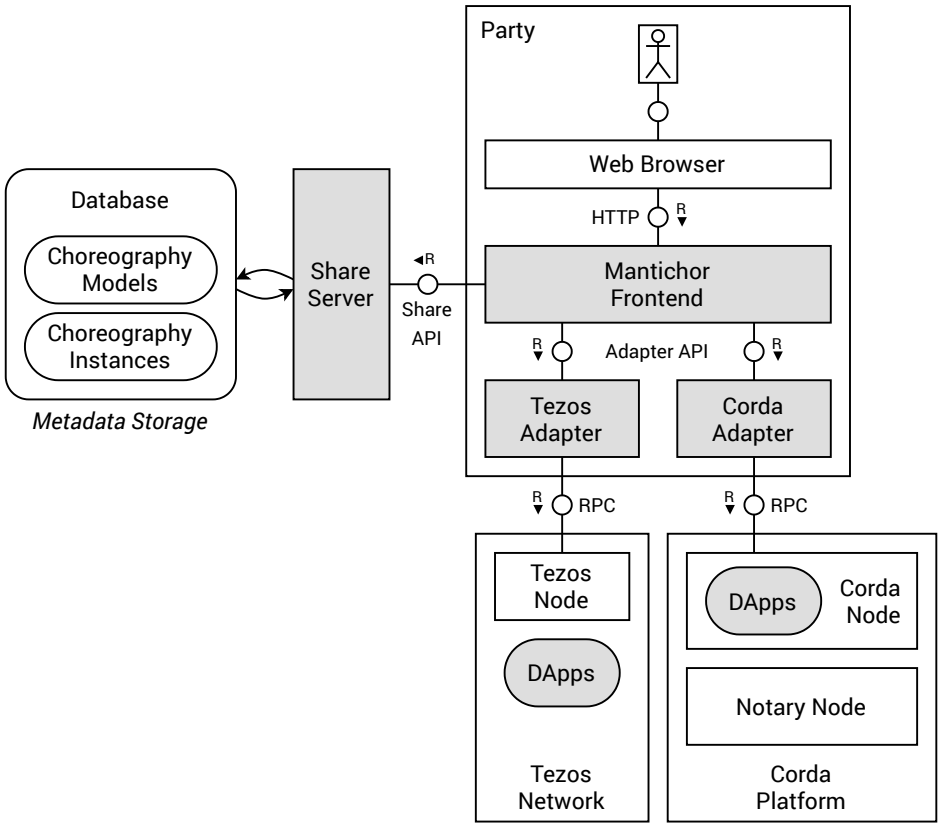


Figure 5.1: Overview of Mantichor's overall system design

Figure 5.2 shows a screenshot of the frontend in the modeling view. The frontend is built with the intention of creating and managing multiple choreography models and several associated choreography instances at the same time, organized in projects. A menu on the left allows users to quickly switch between models—"TT" for train ticket and "R" for a rental contract in the example—, start work on new models, or import models from the share server.

To import a model from the share server, its unique ID needs to be known as described in the next section. Conversely, a model can be shared with others by storing it on the share server. A unique ID is then assigned and supplied via the frontend. At any time, metadata concerning the model can be edited in the model properties modal. In the case of Mantichor, the only model-level metadata is the project name, which is also used to derive the abbreviation shown in the model menu.

Lastly, models can be instantiated and deployed to one of the blockchain networks accessible via the locally available blockchain adapters. Instances are then managed in the execution view, which shows the underlying model of an instance and visualizes the set of enabled tasks alongside the possibility to execute them. Overall, the frontend was designed to be minimal, responsive, and simple.

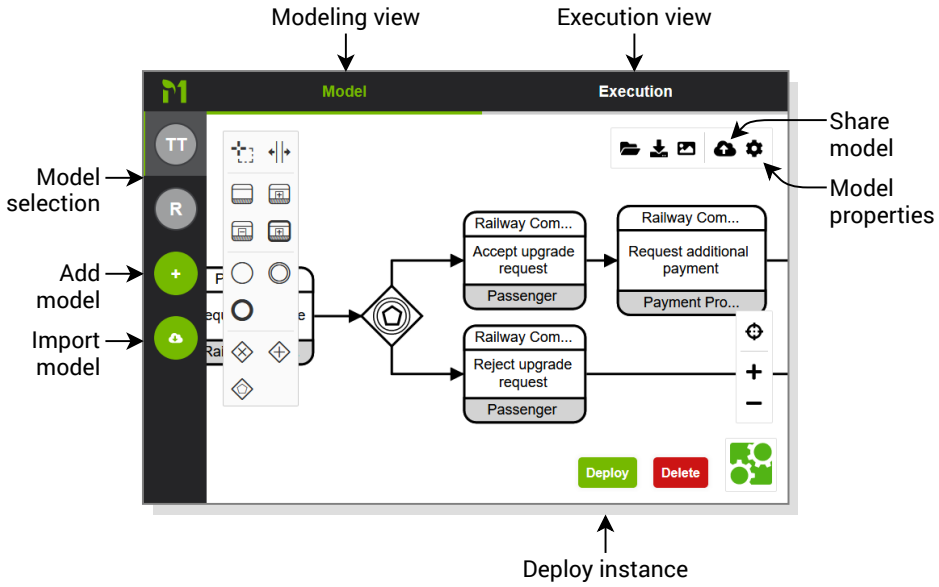


Figure 5.2: Screenshot and components of the Mantichor frontend

5.1.3 Share Server

During all phases of the contract lifecycle, but especially during the initial negotiation and formation phase, participants need to exchange models. The share server is used for just that, implementing a generic and lightweight model, instance, and metadata exchange mechanism. The primary focus as part of the Mantichor project was to enable rapid model exchange, somewhat disregarding authentication and security for simplicity. Like the frontend, the share server was implemented using TypeScript. A MongoDB database instance[2] is used for the actual storage of data and artifacts.

The general sequence of interactions to share a smart contract model between two participants p_1 and p_2 is shown in Fig. 5.3: Participant p_1 sends the model to the share server using the `storeModel` API, allowing arbitrary data as a parameter. The share server uses some method to assign a unique ID to the model, e.g., a hash function. In the case of Mantichor, the result of a hash function was shortened to just a few characters to facilitate a verbal exchange of the ID. The data is stored under this ID in a document-centric database. This ID is returned to p_1, who sends it to p_2 using any medium of communication. Participant p_2 can then query the share server using the ID and the `getModel` API, receiving the original model in the end.

Using this scheme, a group of participants may acquire the same version of a model for negotiation and formation purposes. Checksums can be used to validate the integrity of the model, and that no one tampered with it. In the scope of Mantichor we did not implement any form of authentication when accessing the share server. Instead, security is achieved through the complexity of the ID, making it hard to guess valid IDs as an

[2] `https://www.mongodb.com/`

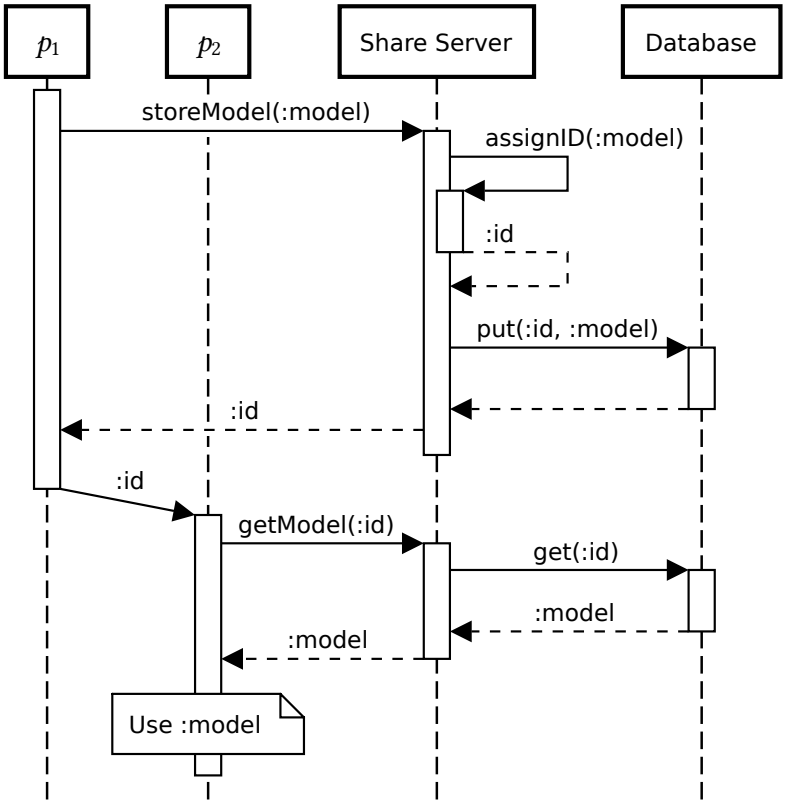


Figure 5.3: Sequence diagram of the interactions of two participants p_1, p_2 to exchange a model using the share server

outsider to the system. It is thus assumed that any person in possession of the ID may also access the associated model. Obviously, more sophisticated protection mechanisms could be conceived, including digital signing and encryption. It may even be advisable for high-stake scenarios to secure the information exchange at this step using blockchain networks already [49].

5.2 Blockchain Network Adapters

At the heart of the SCMS approach are the blockchain adapters. Their main purpose is to be the mediator between the frontend components and the DApps deployed on blockchain networks. They hide the complexity of the blockchain networks and their idiosyncratic inner workings.

5.2.1 Adapter Interface

The Mantichor project was also intended as a first step towards providing a unified interface to otherwise diverse blockchain networks. For this reason, a unified API for all adapters was introduced in Mantichor, which is shown in Tab. 5.1. Each adapter serves a single blockchain network $\mathcal{N} = (N, A)$, and a route `accounts` returns the subset of accounts A the

Table 5.1: Interface of the blockchain network adapters

Verb	*URL and description*
`POST`	`/choreographies` Deploy a new choreography instance **Parameters:** `xml` — Serialization of the choreography model `mappings` — Mapping of roles to blockchain network accounts **Responses:** `id` — Unique choreography instance identifier
`GET`	`/choreographies/{id}/tasks` Query the set of enabled tasks in the given choreography instance **Parameters:** `xml` — Serialization of the choreography model **Responses:** `tasks` — List of enabled tasks
`POST`	`/choreographies/{id}/tasks/execute` Execute a task in the given choreography instance **Parameters:** `xml` — Serialization of the choreography model `task` — Task to execute `address` — Address of the local account to be used
`GET`	`/accounts` Get the active accounts of this adapter **Responses:** `accounts` — List of accounts

adapter may use, i.e., owns the private keys to. In a sense, these accounts are owned by the organization running the adapters and signal the authenticity of all associated transactions and actions within the blockchain network and to the outside world. While in Mantichor those accounts are managed by the adapters themselves, in an enterprise setting a centralized and secured storage within the organization seems appropriate.

The focus of the adapter interface is on the instantiation and execution of choreographies. Three minimal routes were devised for this: one deploying new instances from a choreography model, one returning a list of tasks which are currently enabled, and one executing a task. We designed the adapters to be stateless, which explains why all routes require the original choreography model XML serialization to be provided. This serialization is used to properly format transactions targeting specific flow nodes and also validate the requests. This scheme also requires that the `id` contained in the path of the instance-specific routes must contain enough information to locate the DApp, in our case its address. It would be a simple switch to re-use the IDs assigned by the share server instead for a more consistent experience.

Other than that, the initial instantiation requires a mapping between roles or participants in the choreography model to actual accounts in the blockchain network for authentication purposes. The frontend provides a simple interface for this. Lastly, tasks are identified by their ID within the BPMN choreography model, i.e., the `id` attribute of the `BaseElement` superclass [87, p. 54]. To resolve ambiguities in case the same task is available through multiple call choreographies—a modeling element allowing the reuse of existing model fragments—at the same time, the whole trace of IDs from the root element through the hierarchy of sub and call choreographies needs to be provided.

Both adapters implement a token-based execution semantics as proposed by Weber et al. [109] and refined for the express purpose of BPMN choreography diagrams in our own work [70]. The state of the choreography is characterized by tokens on the sequence flows, and transactions pertaining to a choreography task are rejected if no token can be found in front of it. The set of supported modeling elements is limited to simple event-based and parallel gateways as well as choreography activities.

5.2.2 Tezos Adapter

Tezos is an open-source public blockchain network using a native cryptocurrency called *tez* (XTZ) [38]. The mining and consensus protocols Tezos uses are based on Proof-of-Stake (PoS), in which miners for a block—called bakers—are chosen based on their monetary stake in the procedure. Tezos also has a special feature which allows protocol changes without forking or deploying a new blockchain network: Proposals to change the protocol, e.g., changing gas limits or costs, can be submitted and voted upon by members of the network. If a proposal reaches a certain amount of votes, the protocol changes will become incorporated in the live network.

Listing 5.1: Fi template showing the standard function to execute a task

```
entry ChoreographyTask_[TASK-ID] () {
  assert (SENDER == address "[SENDER]");
  assert (storage.ChoreographyTask_[TASK-ID]_active);
  storage.ChoreographyTask_[TASK-ID]_active =
    bool false;
  storage.ChoreographyTask_[NEXT-ID]_active =
    bool true;
}
```

The Tezos adapter generates DApp code in the *Fi* programming language, which is compiled to Michelson. The generated DApp keeps track of tokens via Boolean flags for each choreography task, sub-choreography, and all sequence flows targeting parallel gateways. Each choreography task is subsequently translated to a function.

A short snippet is given in Listing 5.1, showing a template function for a choreography task. In a first step, the address of the sender of the transaction is checked so that only the participant specified in the diagram can

call the function. The code generator takes into account the role mapping provided in the API (see Tab. 5.1). Then the function checks if the task is really enabled by checking the respective flag. If so, the respective flags of the choreography task and its successor are flipped. If the direct successor is a parallel gateway, the function would also potentially include the semantics of the gateway itself.

5.2.3 Corda Adapter

Corda is a decentralized database which is comparable to a permissioned blockchain, although it internally does not require a strict block structure [45]. Nodes must be granted access to the network, and obtain certifications and keys pertaining to their real-world identity. Corda is different than traditional blockchains like Ethereum or Tezos in that no global broadcast of transactions and DApps is performed. Instead, messages are sent between pairs of nodes in a choreographed fashion called *flow* and exclusively on a so-called "need-to-know" basis, ensuring a level of privacy that is not usually attainable for public blockchains.

The need-to-know principle has a direct consequence, which is that no node has the full ledger available locally. Instead, pieces of information called *facts* encoding the current tokens present in the choreography, are only available to nodes which require and were granted access to this particular information. Two nodes with access to the same fact are guaranteed to have a consistent view on it.

Facts are associated with a notary service part of the network. Any proposed transaction modifying some fact, that is, consuming and issuing a new version of the fact, needs to be notarized by that service. Notarization serves two purposes: For one, double-spending attacks are avoided by the notary keeping track of valid facts which can be consumed. Second, notaries may also validate transactions and check that they conform to the constraints specified in the associated DApp. This allows Corda to work without consensus protocols like PoW [45, Sect. 7.1].

5.3 Insights and Maturity

While working on Mantichor we gained valuable insights into the opportunities but also the challenges of implementing a SCMS.

5.3.1 Functional Coverage

Mantichor covers some of the SCMS features and touches many others, allowing general insights about the full set of features. A summary is shown in Tab. 5.2. Note that we substitute smart contracts for choreographies in Mantichor, which limits the generalizability.

By using chor-js, Mantichor allows users to create fully standard-compliant choreography models. Existing models can be loaded and modified

Table 5.2: Coverage of the SCMS functional requirements in Mantichor

Feature		*Coverage status*
F_1	Create/modify smart contract models	• (Frontend)
F_2	Store smart contract models	• (Share server)
F_3	Share smart contract models	• (Share server)
F_4	Analyze smart contract models	(•) (Frontend)
F_5	Configure privacy/security parameters	○
F_6	Sign smart contract models	○
F_7	Instantiate smart contracts	• (Adapters)
F_8	Share access to smart contracts	• (Share server)
F_9	Implement smart contract semantics	(•) (DApps)
F_{10}	Interact with smart contracts	• (Frontend/Adapters)
F_{11}	Monitor smart contracts	(•) (Frontend/Adapters)
F_{12}	Connect to external oracle services	○
F_{13}	Connect to blockchain interoperability	○
F_{14}	Modify/terminate smart contracts	○
F_{15}	Extract auditing information	○

(F_1). The share server allows the persistent storage of the resulting models (F_2), and also to share them with other participants (F_3). While we do not support advanced analysis of the choreography models, chor-js does provide a validator which performs static analysis. It checks the observability constraints the BPMN standard requires, and makes sure that the choreography model describes an enforceable choreography (F_4).

The adapters are able to deploy choreographies in DApps based on choreography models to their respective blockchain network (F_7). Again, the share server can be used to exchange metadata about the instance, which in particular includes the address or ID (F_8). The DApps implement a small subset of the BPMN choreography diagram standard, being limited to event-based gateways, choreography tasks, and single-level sub-choreography hierarchies. There is no support for autonomous actions in any form, though, limiting the level of support for smart contract semantics (F_9). In Chapter 6, we go into detail as to how autonomous actions may be implemented as well.

The frontend component allows the interaction with deployed choreographies using chor-js: Enabled choreography tasks are highlighted, can be selected, and subsequently executed. The adapter formats and sends an appropriate transaction to the network (F_{10}). This also fulfills the monitoring requirement to some degree, since the current state of choreographies can be viewed at any time, though more elaborate monitoring and performance metrics are missing (F_{11}).

While we did not implement any of the remaining features, none of them would require substantial changes to the architecture of Mantichor. For example, external oracle services (F_{12}) and blockchain-interoperabil-

ity (F_{13}) would be implemented in the existing DApps and adapters, if not a native component of the blockchain network itself [57]. Other features are implicitly given, like extracting auditing information (F_{14}); every transaction to a blockchain network is logged, and is directly attributable through the fundamental non-repudiation property of blockchains. Blockchains have been shown to be susceptible to process mining techniques, further increasing the potential for after-the-fact auditing [62].

For long-running smart contracts, blockchain migration may be another issue. When blockchain networks reach their capacity or are superseded by newer versions—see, for example, the sequence of test networks used for Ethereum—DApps may be prematurely deactivated or migrated to new blockchain networks. We did not consider such functionality in Mantichor, but appropriate patterns may be introduced in adapters and DApps readily [13].

5.3.2 Non-Functional Properties

Developing Mantichor provided insights into to the non-functional properties of SCMS implementations as well. To structure the following discussion, we will refer to a subset of the ISO 25010:2011 system and software quality requirements as posed by the International Organization for Standardization (ISO) [52].

Performance Efficiency

The performance efficiency metric expresses the performance, i.e., the processing times and throughput, of a system taking into account the resources consumed. We found that in the case of Mantichor, the blockchain networks and interactions with them were the dominant factors for performance and resource usage, and that the minimal frontend and share server components did not exhibit any noticeable performance issues.

The local Tezos and Corda nodes set up in our tests put a major strain on the test systems—all regular consumer hardware—, both regarding the initial startup and setup phase as well as the inclusion delay of any transaction. In practice, of course, adapters may not need to run their own blockchain nodes, but use external services like Infura (see Sect. 4.3.3) to use a gateway to the blockchain network.

The perceived performance especially suffers from the inclusion time, that is, the delay between submitting a transaction (executing a choreography task) and it being confirmed by the blockchain network. While we tried to clearly communicate these delays to users through the frontend, e.g., by having modal loading screens and messages, the system as a whole often feels unresponsive. Blockchain networks are notoriously hard to scale [118], so this might be a big factor when deploying larger and more complex DApps than we did for Mantichor.

Compatibility

The SCMS architecture is component-based, and components were implemented as services. That is, many of the services may co-exist on the same system without them running into compatibility issues. In fact, Mantichor was mostly developed and run on individual systems during development, highlighting the benefits of this architecture.

Interoperability between multiple Mantichor installations, e.g., at different organizations, is achieved via the share server and also through the blockchain networks. There is no provision for Mantichor installations to communicate beyond those means, as this would imply interactions beyond the scope of the contract lifecycle and specification and introduce a potential for collusion.

Reliability

We believe that the reliability of the overall SCMS approach heavily relies on that of the blockchain networks used. That is, reliability [74] and availability measures [110] as applied to blockchain applications and networks, respectively, also apply to the SCMS approach. In particular, uncertainties regarding the eventual inclusion or dismissal of transactions need to be addressed in implementations [117].

In this context, blockchain networks also provide great benefits. For one, availability is generally very high, since there is no single point of failure and larger networks are quite robust. More importantly, the storage of the entire smart contract state in the DApps guarantees a high degree of recoverability for all sensitive and critical information.

Security

In Mantichor, we relaxed security considerations in favor of more rapid development. As such, Mantichor certainly does not meet security requirements as would be expected in practice, like the missing authentication measures for the share server. However, the SCMS architecture and the use of blockchain technology does, by default, benefit security metrics.

In particular, if private or consortium blockchain networks like Corda are targeted, a high degree of confidentiality can be achieved. The fundamental properties of blockchain—integrity, non-repudiation, accountability, and authenticity [115]—are also directly transferred to the operational logic of the choreography contained within the DApps. As such, even if we did not particularly strive for security, Mantichor highlights the opportunities in these regards.

Maintainability

The act of maintaining Mantichor turned out to be one of the major undertakings during its development due to the ever-changing nature of blockchain technology. Tezos and Corda would receive updates fairly regularly,

often solving some issues we were facing but also containing breaking changes or new features. In general, we feel like this is a major issue and uncertainty when developing against blockchain technology as of now, both for Mantichor and for other software systems using it.

Regarding maintainability, the immutability property of DApps certainly comes into play. Once deployed, the code of a DApps remains static, and only its state changes. As a consequence, retroactively changing parts of the DApp is generally not possible by design. As discussed in Sect. 4.2, patterns exist to lift this restriction for selected parts of DApps [61], but more clerical mistakes in other parts may not be covered by the patterns. This is a major concern for smart contracts, since faulty DApps could lead to incorrect behavior and disputes.

Portability

Again, the immutability of DApps introduces issues when it comes to their portability. Different blockchain networks may work radically different and require individual adapters. The logic used in one adapter is not easily transferred to another and vice-versa. Like for the maintainability metrics, portability also depends on blockchain migration techniques in the future [13, 61].

Usability

The experience a user has with a system is paramount for their satisfaction and the overall results. The major challenge when it comes to usability in the context of blockchain-based applications is the delay with which transactions are safely included in a blockchain (see the transaction lifecycle in Sect. 2.2.3). Applications and thus any implementation of SCMS needs to be able to communicate any delays appropriately, and make sure the user does not perceive the system as being slow or unresponsive. Other than that, usability was not one of the major goals of Mantichor outside of providing a minimal, simple, and responsive application.

5.3.3 Current Status

During development, the students would often encounter technical difficulties related to the blockchain networks employed. Corda and Tezos are complex from both conceptual and infrastructure perspectives, and frequent updates considerably changed techniques and interfaces. Thus, the originally planned set of features could not be finished in scope of the seminar.

The Tezos adapter was implemented and tested using a local, sandboxed network running the Alphanet protocols. The sandbox mode limited the use of several features network features like libraries, which could be used to improve the implementation when used against a live or testing network.

A particular issue in the case of Corda was the poor performance of our DApp generation and deployment approach, which generated a large amount of boilerplate code for every choreography. On regular consumer hardware (MacBook Pro, 2012), compiling the DApps and embedding them in the network nodes would take upward of 10 minutes, and nearly twice as long when using the dockerized variant. While this may certainly be changed with more sophisticated code generation and deployment strategies, it made the Corda adapter impractical to use in any realistic setting.

For the reasons mentioned above, Mantichor was not pursued further within the scope of this thesis. While originally planned to be a basis for future extensions and other prototypes, the dynamic and ever-changing world of blockchain technology as well as the considerable amount of work involved in just maintaining the existing set of features rendered it not worthwhile from a scientific perspective. We will further discuss the implications of this decision in Chapter 9 of this thesis, especially regarding the perceived feasibility in practice.

5.4 Modeling Choreographies with chor-js

In Mantichor, we used BPMN choreography diagrams to express smart contracts which are subsequently enforced using DApps. To this end, we integrated *chor-js*, a web-based BPMN choreography diagram editor based on the bpmn-js[3] project by bpmn.io and Camunda [69].

chor-js is fully open-source and maintained publicly on GitHub[4] with the help of Anton von Weltzien, and has since gained some popularity in research and from tool vendors. As such, we consider chor-js itself to be a major contribution of this thesis, which we briefly wish to outline.

5.4.1 Feature Overview

BPMN choreography diagrams contain some of the most complex atomic modeling elements in the standard in the form of choreography activities, which reference and visualize multiple participants and messages using participant bands and message decorators, respectively [112]. For chor-js, we implemented several features specifically designed and tailored to handle their specification and peculiar syntax:

- **Participant bands** show which participants are involved in a choreography activity and who is the initiator of a choreography task via their shading. The BPMN standard defines clear rules on how those participant bands may be attached and ordered [87, Tab. 12.33]. These rules are faithfully implemented in chor-js, and features to move participant bands and switch the initiators are available.

[3] `https://github.com/bpmn-io/bpmn-js`
[4] `https://github.com/bptlab/chor-js`

- Any interaction between participants is further signified by a **message** attached to a participant band. Messages can only be attached to choreography tasks, which may reference up to two of them; one for the request message, and one for the response message. Both message decorators may be hidden or shown in the diagram at the discretion of the model designer, which chor-js readily supports.
- Both means of **hierarchical modeling**, sub-choreographies and call choreographies, are fully supported in chor-js. For the latter, we also support the creation of multiple diagrams in the same file, after which they can be interlinked and participants can be mapped between diagrams.
- A **validation tool** built on top of chor-js provides static analysis capabilities of several constraints put forward in the BPMN standard. This particularly includes the initiator rule, violating which may lead to non-enforceable models, the completeness of participant mappings when using call choreographies, and more.

We fully implemented the BPMN standard's diagram interchange XML format for choreography diagrams in chor-js. This enables interoperability between other tools working with BPMN choreography models.

5.4.2 Tool Comparison

Several BPMN tooling vendors provide some support for BPMN choreography diagrams, albeit often severely lacking or incomplete in several aspects. An overview is shown in Tab. 5.3 where a checkmark means (close to) full and native support, a cross means no support, and a bullet means no or broken support. Most code bases of commercial vendors like Signavio or Visual Paradigm are closed to the public, making a full assessment difficult.

Still, chor-js outperforms all other offerings in most aspects, only lacking support for choreographies between collaborations—a combination with BPMN collaboration diagrams—, boundary events for the reasons described in the next section, and global choreography tasks for their insufficient specification in the standard. Due to chor-js being an open source project, support for these missing features may be added in the future.

5.4.3 Scientific Contribution

While working on chor-js, several problems concerning the model and semantics of BPMN choreographies became apparent. For example, it is not clear how the initiator rule saying that an initiator of a choreography activity must have been involved in some capacity in the previous choreography activity translates to sub-choreographies, since involvement in the sub-choreography does not mean a participant is aware of the tasks at

Table 5.3: Tool comparison between chor-js and selected competitors, reproduced from [69]

			chor-js	Signavio	BPMN2 Modeler	Visual Paradigm	Trisotech
features	code base		open	closed	open	closed	closed
	XML diagram interchange		✓	•	✓	•	✓
	standalone diagrams		✓	✓	✓	✓	✓
	between collaboration		×	×	×	•	•
element support	**gateways**		✓	✓	✓	✓	✓
	events	start, end, intermediate	✓	✓	✓	✓	✓
		boundary	×	•	×	×	•
	messages (attached)		✓	✓	✓	×	✓
	choreo. activities	choreo. task	✓	✓	✓	✓	✓
		sub-choreo.	✓	✓	•	•	✓
		call choreo.	✓	•	•	•	×
		global choreo. task	×	×	•	×	×
	markers	loop type	✓	✓	✓	✓	✓
		participant multiplicity	✓	×	✓	×	✓

its end. Further, the modeling and referencing of data contained in messages, which may be used in data-based exclusive gateways, is not clearly stated. The issue of determining which data is visible to which participant is only vaguely discussed.

Perhaps most importantly, the BPMN standard fails to properly explain the semantics of boundary events attached to choreography tasks, even omitting them in the metamodel. More on these and other issues we faced during the development of chor-js is reported in an accompanying blog post[5] co-authored by the author of this thesis. While the BPMN standard seems to be relatively static since its newest version has been released in January of 2014, these insights might help to make BPMN choreography models more complete and sound in the future.

In summary, we hope that chor-js enables researchers to use BPMN choreography models and diagrams in their research—not least since the discipline of choreography modeling "may be re-vitalized" [83] by blockchain technology.

[5] `https://camunda.com/blog/2021/01/chor-js-an-editor-for-bpmn-choreography-diagrams/`

Chapter 6

Autonomous Actions on Blockchain

An important building block of smart contracts are autonomous actions, which are executed automatically and without delay as soon as they become enabled. They do not belong to an individual party but to the smart contract as a whole, elevating it to take an active part in the contractual proceedings. No party should be able to influence the correct and timely performance of autonomous actions, lest it could be used to manipulate the outcome of the smart contract—either by changing, delaying or even completely avoiding their execution.

Ostensibly, the SCMS approach takes care of this by encoding the smart contract within immutable and tamper-proof DApps, which include provisions to execute autonomous actions. However, owing to its non-continuity property (see Sect. 2.2.4), the DApp still needs to be triggered by transactions to do that, and sending a transaction is an inherently intentional act by some entity outside of the blockchain network. Whether this entity is a party, an oracle provider, or some other software component; there can be no guarantee that a sufficient amount of transactions is sent for the timely performance of all autonomous actions. The DApp must somehow mitigate any influence this has on smart contract enforcement.

In this chapter, we will first outline the issues in detail (see Sect. 6.1), before devising two approaches to resolve them (see Sect. 6.2). The oracle patterns necessary for these approaches are introduced in Sect. 6.3 and implemented in Sect. 6.4. We use the implementation to assess the feasibility of our approach in Sect. 6.5.

Parts of this chapter are based on our previous work, in which we explored the issues regarding external data monitoring from within the restricted environment of DApps [72]. A significantly extended version of that work which is also included in this chapter is currently in manuscript stage [68].

6.1 Performing Autonomous Actions

In a first step, the DApp needs to determine whether an autonomous action is enabled in the first place, which depends on various criteria. In the following, we identify a notion of external events encapsulating the problematic of these criteria.

For illustration purposes throughout this chapter, we will consider the smart contract model $\mathcal{M}_{ticket} = (P, D, L, A, C)$ from Sect. 3.2.1 (see page 39) modeling the train ticket smart contract, and further assume it is implemented using a single DApp Ð. We refer with $a_c := a_4 \in A$ to the autonomous action canceling the ticket in case the passenger's discount card expires.

6.1.1 Enablement Criteria

Whether an action is enabled in general depends on the smart contract state $s = (\Lambda, v, \mathrm{H})$ and the current operating environment state $o = (t, \phi)$. In Tab. 6.1 we summarize the criteria of enablement for an action a, that is, (i) the presence of the required legal relations $\mathsf{Pre}(a)$ in the current smart contract state as well as (ii) the satisfaction of all performance constraints attached to a (see Sect. 3.2).

In the case of a_c, only one legal relation needs to be in the current smart contract state: l_1, which permits the passenger to use the train. This is trivial for Ð to check, since it stores and maintains the smart contract state s, including Λ, itself. Thus, Λ only changes after a state transition within a transaction, and allows Ð to immediately react and execute any newly enabled autonomous actions. Indeed, this is the case for all other criteria depending on the smart contract state s, like causal constraints evaluating the history of past actions H or data constraints depending on the valuation v of internal data sources.

However, data and temporal constraints often depend on the state of the operating environment—which may change outside of transactions. In the case of a_c, the attached temporal constraint $c_c \in C^T$ pertains to the expiry of the passenger's discount card, which is tied to the system time

Table 6.1: Influence of smart contract state and operating environment state on the enablement of actions

Criterion		*Affected by state of...*	
		Smart contract $s = (\Lambda, v, \mathrm{H})$	*Operating environment* $o = (t, \phi)$
Legal relations		✓ Λ	—
Performance constraints	Causal	✓ H	—
	Data	✓ $v(d)$ for $d \in D^P \cup D^V$	✓ $\phi(d)$ for $d \in D^E$
	Temporal	✓ H (activation time)	✓ t

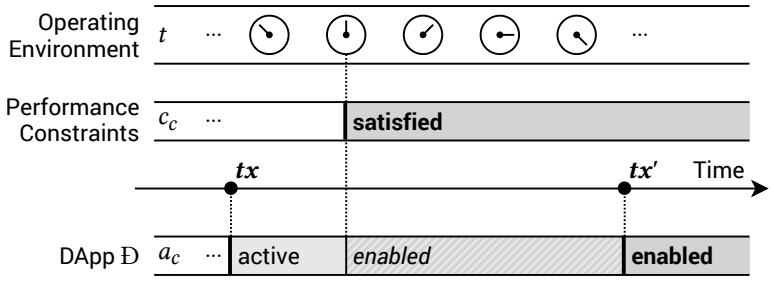


Figure 6.1: Enablement of a_c due to a change of the operating environment state

t of the current operating environment state o. As shown in Fig. 6.1, it may happen that a_c becomes activated in a transaction tx, but its later enablement due to the performance constraint c_c becoming satisfied is only detected with a delay once a transaction tx' arrives. As a result, a_c will be performed too late, violating the operational semantics of smart contracts.

6.1.2 External Events

To narrow down the challenges of noticing an action's enablement in the first place, we will consider the role of *events* as its driver. The idea is that an action which becomes enabled within a transaction is immediately performed, potentially leaving only those which are active and blocked by a non-satisfied performance constraint. The performance constraint then may become satisfied once an event occurs: the change of the valuation of some external data source for data constraints or the passing of a deadline for temporal constraints. Using this perspective, the task of noticing an autonomous action becoming enabled is subsumed in that of detecting an event. In the following, we will assume for simplicity that each action has exactly one such event attached to it, and the event is specified by a single performance constraint.

As we have observed above, only those events originating from the operating environment state o are problematic for the DApp Đ since they happen outside of transactions. For example, the action a_c canceling the train ticket becomes enabled due to the event of the passenger's discount card expiring, as specified by the temporal performance constraint c_c. We call these events *external* since they occur outside the influence of the smart contract:

Definition 31 *(External Event).* Let $\mathcal{M} = (P, D, L, A, C)$ be a smart contract model. Further, let $s \in \mathbb{S}$ be a smart contract state, and $o, o' \in \mathbb{O}$ with $o \longrightarrow o'$ be two consecutive operating environment states.

Then an *external event* with respect to an autonomous action $a \in A$ occurs whenever the transition from o to o' causes a to become enabled, that is, $\sigma(a, s, o) = \mathsf{active}$ and $\sigma(a, s, o') = \mathsf{enabled}$. ◇

When such an external event occurs, the autonomous action a must immediately be performed by the DApp Ð—that is, Ð needs to detect and act upon the event promptly. Before that, however, a second issue comes into play, which is that of competing actions.

6.1.3 Competing Actions

Autonomously canceling the train ticket once the passenger's discount card expires using a_c is not the only autonomous action in the smart contract model $\mathcal{M}_{ticket}$. In fact, there are two more such actions which may be active or enabled at the same time:

- $a_d := a_1$, which registers the *departure of the train* and validates the ticket. It is constrained by c_d, a data constraint accessing an external data source d_d, i.e., a train departure service managed by the RIM. a_d may thus be enabled by an external event, the train's departure as communicated by the organization managing the railway network as a whole.

- $a_w := a_2$, which registers a *severe weather warning* and entails the refund of the ticket as a voucher. It is constrained by c_w, a data constraint accessing an external data source d_w, i.e., a weather warning service maintained by an approved meteorological institution. a_w may thus be enabled by an external event, the issuance of a severe weather warning.

The actions a_c, a_d, and a_w compete in a race against each other: They are active in every initial state of the smart contract, and whichever one is enabled first must also be performed first by the smart contract. This race, of course, may also be joined by non-autonomous actions; in this case $a_t := a_5$ canceling the ticket by the choice of the passenger.

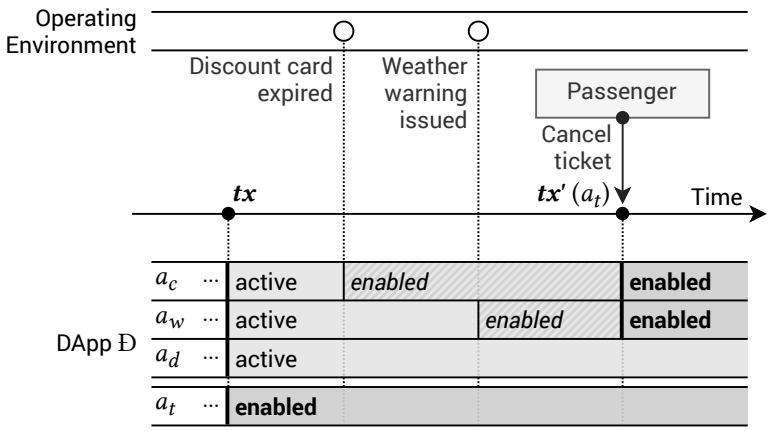


Figure 6.2: A race between competing actions which needs to be resolved by the DApp

The winner of the race can only be determined by the DApp Ð in a transaction. Figure 6.2 shows a timeline view of how such a race may play out, with the operating environment abstracted to just its external event occurrences. In a first transaction tx, the smart contract state changes such that the autonomous actions a_c, a_w, and a_d are activated, and a_t—choice of the passenger—is enabled. None of the autonomous actions are enabled since their performance constraints are not yet satisfied.

After tx has finished, the discount card status expires and a_c becomes enabled, after which a severe weather warning is issued and a_w becomes enabled. Both of these external events are not detected yet by Ð, however, as it lies dormant and is only called again in a transaction tx' associated with action a_t—in essence, the passenger trying to cancel their ticket. Of course, the only correct state transition would be to perform the autonomous action a_c, since it was enabled first. There is no way for Ð to know this is the case, though, since it has no knowledge about when exactly an external event happened. As such, specific strategies are needed to resolve this uncertainty and correctly enforce smart contracts in DApps.

Overall, this situation is very similar to the deferred choice pattern in workflow modeling [96]. Deferred choice refers to situations in which there is an exclusive choice between several execution branches, each associated with some event caused by circumstances outside the influence of the workflow engine. Only one of these branches is picked based on which event is detected first, essentially modeling a "race condition where the first [e]vent that is triggered wins" [87, p. 298]. However, in the context of smart contracts we do not necessarily require the competing actions to be exclusive to each other. Rather, there can also be situations in which the autonomous actions complement each other, but still need to be executed in the correct order. As such, the problem stated above subsumes deferred choice, but takes on a more global perspective.

6.2 Event Detection Approaches

Correctly implementing autonomous actions within DApps comes down to detecting external events, and resolving associated ordering conflicts. In this section, we propose two novel solutions to these issues; first, by retroactively detecting events and postdating the performance of the autonomous actions in the correct order, and second by making sure that transactions are sent whenever an event may be detected.

6.2.1 Retroactive Event Detection

Since the core problem discussed above is that of not detecting events, we propose an approach to retroactively detect events and resolve any resulting conflicts, i.e., making sure that the order of actions as prescribed by the operational semantics is honored. For this, we need additional capabilities, especially regarding the operating environment.

Enablement Time

The core information necessary to decide if, and which, autonomous action should be performed is the state of the operating environment. Since the operating environment is in an ever-changing, transient state, DApps do not normally have access to past valuations of external data sources or timestamps. For the retroactive event detection approach, however, we assume that those historical operating environment states are available. We can then determine when exactly an autonomous action first became enabled, which in turn also marks the initial occurrence of the associated external event. For completeness reasons, we also need a way to express that an event never became enabled, which we will do using the future timestamp:

Definition 32 *(Future Timestamp).* The timestamp $\top \in \mathbb{N}$ represents a point in time sufficiently far into the future so that it will never be reached by any practically observed timestamp $t \in \mathbb{N}$, i.e., $t \ll \top$. ◇

The future timestamp essentially serves as a unique flag. We decided against adding an additional element to the time domain $\mathbb{N}$ (see Def. 6 on page 14) for this purpose, since it breaks the basic arithmetic properties of the time domain and unnecessarily complicates our specification. Further, implementations thus do not require complex data structures to represent timestamps (see Sect. 6.8). Then, the enablement time is defined as follows:

Definition 33 *(Enablement Time).* Let $\mathcal{M} = (P, D, L, A, C)$ be a smart contract model with an instance $\mathcal{S}$ in a state $s \in \mathbb{S}$, and $o = (t, \phi)$ be the current operating environment state. Further, let $o_0 = (t_0, \phi_0)$, ..., $o_n = (t_n, \phi_n) = o$ be the consecutive operating environment states from when $\mathcal{S}$ entered the state s, i.e., the last transaction.

Then the *enablement time* t_{enabled} of an action $a \in A$ is the timestamp at which a was first enabled since o_0 until o_n, more formally defined as

$$t_{\mathsf{enabled}}(a, s, o) := \begin{cases} \top, \text{ if } \sigma(a, s, o) = \mathsf{disabled} \\ \min\left(\{\top\} \cup \{t_i \mid i \in [0, n] \wedge \sigma(a, s, o_i) = \mathsf{enabled}\}\right), \\ \quad \text{otherwise} \end{cases}$$

◇

The enablement time $t_{\mathsf{enabled}}(a)$ of an action $a \in A$ specifies when a first could have been performed since the last transaction, and *should* have been in the case of autonomous actions.

Conflict Resolution

The enablement time can now be used to perform autonomous actions retroactively in the correct order, meaning that any conflicts of competing actions will be resolved. In Fig. 6.3 we show a pseudocode algorithm

```
1  Transaction Perform(a ∈ A):
2    s ← Current smart contract state (Λ, v, H) ∈ 𝕊
3    o ← Current operating environment state (t, φ) ∈ 𝕆
4    s ← PerformAutonomousActions(s, o)
5    If σ(a, s, o) = enabled:
6      s ← New smart contract state s′ after performing a, i.e., s --o/a--> s′
7      s ← PerformAutonomousActions(s, o)
8    Endif
9    Set s as the new smart contract state

10 Local Function PerformAutonomousActions(s ∈ 𝕊, o ∈ 𝕆):
11   Do:
12     Ã ← {a′ ∈ A | ChoiceOf(a′) = ∅ ∧ t_enabled(a′, s, o) ≠ ⊤}
13     If Ã ≠ ∅:
14       ã ← Pick a′ ∈ Ã such that t_enabled(a′, s, o) is minimal
15       õ ← Operating environment state at t_enabled(ã, s, o)
16       s ← New smart contract state s′ after performing ã, i.e., s --õ/ã--> s′
17     Endif
18   While Ã ≠ ∅;
19   Return s
```

Figure 6.3: Pseudocode of the retroactive event detection approach

outlining the implementation of the approach, abstracting from the actual interface of a DApp to a generic function `Perform` (line 1) describing a transaction. This function is called with a parameter $a \in A$, designating the original "target" action of the transaction—i.e., the usually non-autonomous action which some participant wishes to execute and sent the transaction for. The target action can also be autonomous if a party wishes to expedite the performance of a, or if the smart contract only consists of autonomous actions to begin with. Every transaction has a target action.

In lines 2–3, the current smart contract state s as well as the current operating environment state o are gathered. Before even considering a, though, a local function is called that performs all autonomous actions which are currently enabled (line 4). The resulting new state is stored, and only if a is enabled under these new circumstances (line 5) is it finally performed (line 6). As this may lead to yet another set of autonomous actions becoming enabled, the local function is executed again (line 7). Only then is the smart contract state updated to reflect the performed actions (line 9).

The local function `PerformAutonomousActions` (line 10) expects a smart contract state s and an operating environment state o as an input. It starts by calculating the set $\tilde{A}$ of autonomous actions which should have been performed already given the state information (line 12). If this set is not empty (line 13), then the autonomous action $\tilde{a}$ which was enabled first (line 14) is picked from the set. Note that there might be multiple actions with the same earliest enablement time, and the algorithm has to make an arbitrary but fixed choice in accordance with the non-determinism of the

smart contract semantics (see Sect. 3.2). In line 15, the operating environment state $\tilde{o}$ which held when $\tilde{a}$ should have been performed is acquired. The autonomous action $\tilde{a}$ is then performed, backdated to $\tilde{o}$, leading to a new smart contract state (line 16).

The procedure is repeated until no autonomous action may be performed anymore. The loop structure is necessary since the performance of one autonomous action may lead to the initial activation and enablement of previously disabled actions and vice-versa. For example, a sequence of autonomous actions with relative temporal constraints keeping them a few minutes apart may all be executed one after the other in a single transaction.

A noteworthy consequence of this algorithm is that the action a, the original target of the transaction, may not even be executed. It is the task of the algorithm to make sure that the autonomous actions are performed in the right order, but also that no action which is not enabled is performed—which also applies to a. Parties must be aware that the transactions they send may include such seemingly unrelated activities, and may have unexpected yet well-defined outcomes.

Of course, the historical states of the operating environment are not available to a DApp by default. Thus, there must be some entity to keep track and provide those states on demand for the enablement time to be calculated within the DApp. To this end, we will propose conservative extensions to the existing oracle patterns in the next section.

6.2.2 Publish-Subscribe Event Detection

The idea of the *publish-subscribe event detection approach* is to avoid having to do retroactive conflict resolution and backdating. Instead, there must be an assurance that whenever an external event may have occurred in the operating environment, the DApp Ð is triggered by a transaction. While this might lead to superfluous transactions, it assures that no external event is missed. Within these transactions, Ð could thus be sure that no action became enabled before another and should take precedence—assuming the DApp is using a monotonic time measure and no transaction with an earlier timestamp may arrive later (see Sect. 7.2.7).

The pseudocode algorithm for the publish-subscribe event detection approach is shown in Fig. 6.4 and is very similar to the retroactive approach, but notably removes the requirement for knowing any past operating environment states. A transaction may now target no specific action, *nil*, at all (line 1), the only purpose being to check the enablement of autonomous actions since something *may* have happened in the operating environment. Again, all enabled autonomous actions are performed before any preferred action a given as a parameter (line 4). This time, however, the local function (lines 10–19) only checks for enablement in the current operating environment state. Again, there is a non-deterministic compo-

```
1  Transaction Perform(a ∈ A ∪ {nil}):
2    s ← Current smart contract state (Λ, ν, H) ∈ 𝕊
3    o ← Current operating environment state (t, ϕ) ∈ 𝕆
4    s ← PerformAutonomousActions(s, o)
5    If a ≠ nil ∧ σ(a, s, o) = enabled:
6      s ← New smart contract state s′ after performing a, i.e., s —o/a→ s′
7      s ← PerformAutonomousActions(s, o)
8    Endif
9    Set s as the new smart contract state

10 Local Function PerformAutonomousActions(s ∈ 𝕊, o ∈ 𝕆):
11   Do:
12     Ã ← {a′ ∈ A | ChoiceOf(a′) = ∅ ∧ σ(a′, s, o) = enabled}
13     If Ã ≠ ∅:
14       ã ← Pick any from Ã
15       s ← New smart contract state s′ after performing ã, i.e., s —o/ã→ s′
16     Endif
17   While Ã ≠ ∅;
18   Return s
```

Figure 6.4: Pseudocode of the publish-subscribe event detection approach

nent in this, as the algorithm just picks an arbitrary-but-fixed action if more than one is enabled.

Such an approach would require an off-chain component, e.g., as part of the local SCMS components of the parties' SCMSs, which observes the operating environment and automatically triggers the sending of transactions in case relevant changes occur. The consideration of what constitutes a relevant change would be up to the implementation. For a data constraint, a transaction could be sent whenever the valuation ϕ changes at all or only when the change leads to the constraint becoming satisfied. This may introduce trust issues, since it is possible that participants would deliberately delay or refrain from sending such transactions selectively.

However, instead of introducing an entirely new component for this purpose, we opted for extending existing oracle patterns. Events can only happen when the operating environment changes, and oracles are the gateway to the operating environment. Thus, oracles can be extended to handle these responsibilities as well, and notify interested DApps whenever a valuation changes.

6.3 Extended Oracle Architectures

To access the operating environment state, particularly the valuation of the external data sources, from within a DApp, oracles (see Sect. 2.2.5) are needed. This also applies to the two event detection approaches introduced in the previous section.

In the following, we assume that each external data source $d \in D^E$ of a smart contract $\mathcal{M} = (P, D, L, A, C)$ is associated with exactly one unique oracle, and that the smart contract DApps are the consumers of these oracles. Further, we assume that oracles are only used for data constraints, and that temporal constraints can be evaluated using timing information of the blockchain network available to the DApp. We will discuss the issues concerning the availability of timing information in Chapter 7.

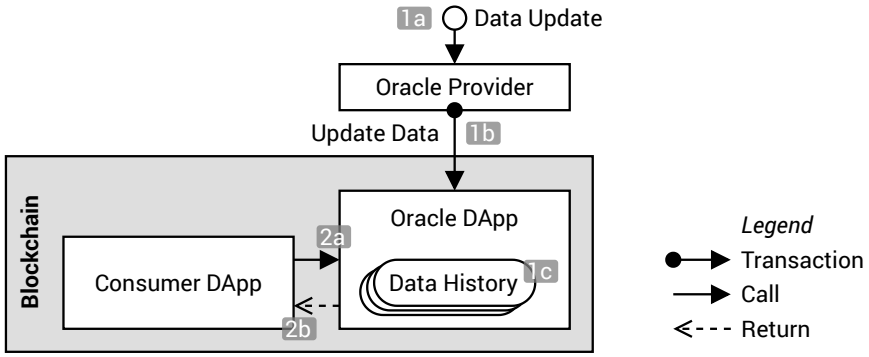
The two most common oracle patterns, storage and request-response oracles (see Sect. 2.2.5), can only be used to acquire the current operating environment state and thus only the current valuation of d, however [116, 84]. They lack the capability to provide historical information, or actively notify the DApp of changes to the valuation of d which might enable an autonomous action. To support the two event detection approaches, we thus propose a set of purposeful extensions and refinements of the existing architectures precisely targeting the required capabilities.

6.3.1 History Oracles

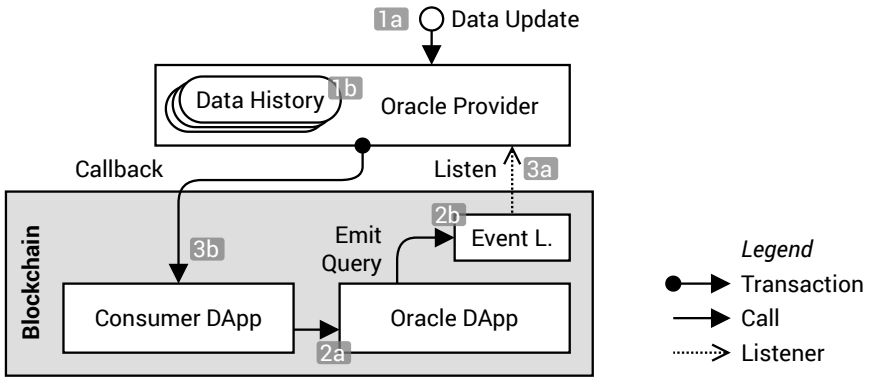
History oracles can be used to acquire the historical valuations of external data sources, essentially allowing a consumer DApp to reconstruct a sequence of past operating environment states. This enables the calculation of the enablement time of actions which is needed to implement the retroactive event detection approach. We introduce two variants of the history oracle:

The on-chain history oracle (see Fig. 6.5a) extends the storage oracle pattern, which allows synchronous access to external data by storing it in a DApp. To this end, the oracle provider observes the external data source using any means available to it, and on updates acquires the current value (1a) and sends a corresponding transaction to the oracle DApp (1b). The oracle DApp maintains a list of timestamped values of the data source, including the previous values, to which the new information is appended (1c). Consumer DApps can then request an arbitrary slice of this current and past data (2a), which is immediately returned (2b).

The off-chain history oracle (see Fig. 6.5b), on the other hand, is based on the existing request-response oracle pattern. Again, the oracle provider observes the external data source, but this time stores any updated values (1a) locally outside the blockchain network in some list or database (1b). When a consumer DApp requests a slice of the current and past data (2a), this request is not immediately answered but emitted using the blockchain's event layer (2b). The oracle provider observes this event (3a), locally prepares a new transaction containing the response data, and directly targets this to the consumer DApp (3b).



(a) Synchronous on-chain history oracle



(b) Asynchronous off-chain history oracle

Figure 6.5: Architecture and behavior of the history oracles

6.3.2 Publish-Subscribe Oracles

The history oracle patterns both need to observe the external data source constantly to acquire any updates promptly. While we do not prescribe any particular technique with which this is done, in practice an observer or publish-subscribe software pattern is often used: The oracle provider registers as a subscriber at the external data source or an intermediate, which then actively notifies them of any data update.

The idea of the publish-subscribe oracle is to extend this pattern to the consumer DApp. That is, the oracle provider serves as the intermediary between the provider of the data, the external data source, and the subscriber to the data, the consumer DApp. This way, the consumer DApp would be kept in the loop about the value of the external data source, and no large slices of historical data would have to be transferred. This directly corresponds to the publish-subscribe event detection approach which we have introduced in the previous section.

Figure 6.6 shows a resulting architecture of such a publish-subscribe oracle. Again, the oracle provider notices or is notified about data updates (1a), and stores a cached version of the current value locally (1b). At the same time, already subscribed consumer DApps are provided with the current value with a transaction each (1c). To newly subscribe to the oracle, a

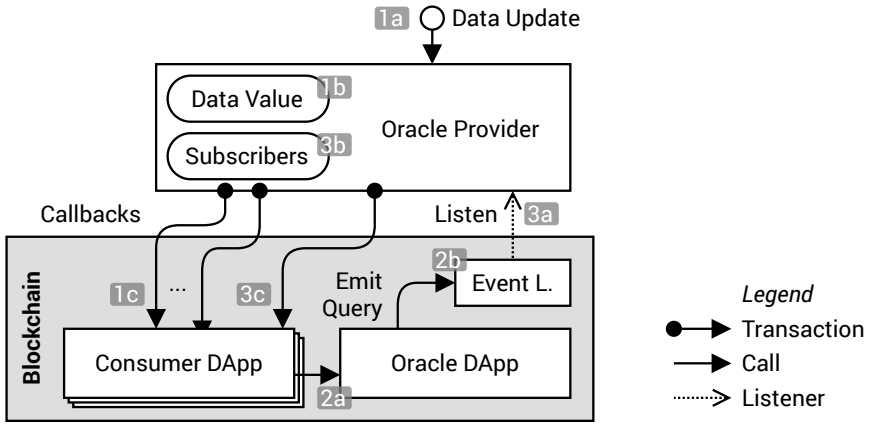


Figure 6.6: Architecture and behavior of the publish-subscribe oracle

consumer DApp queries the associated oracle DApp (2a), which emits an appropriate event using the blockchain's event layer (2b). This is picked up by the oracle provider (3a), who adds the new subscriber to its list of subscribers (3b). Future data updates will then be communicated to the consumer DApp, and to avoid any gaps of knowledge the cached current data value is immediately sent to them via a transaction (3c).

6.3.3 Conditional Oracle Variants

In Tab. 6.2 we summarize the interfaces of the oracle architectures from the perspective of the consumer DApp, that is, which parameters need to be attached to a query, and which format do the results arrive in. The storage and request-response oracle architectures, for instance, require no additional input parameters—assuming again that each oracle is statically associated with exactly one external data source—and provide the consumer DApp with a singular value from the data domain $\mathbb{D}$. The publish-subscribe oracle exposes the same interface, but there may be more than one callback depending on the evolution of the external data source. The history oracle, on the other hand, requires consumers to provide a timestamp

Table 6.2: Interfaces of the oracles from the perspective of the consumer DApp

Variant	*Oracle pattern*	*Interface domains*	
		Parameters	*Result*
Regular	Storage, req/res	*none*	$\mathbb{D}$
	History	$\mathbb{N}$	$(\mathbb{N} \times \mathbb{D})^*$
	Publish-subscribe	*none*	$\mathbb{D}$ (repeated)
Conditional	Storage, req/res	EXPR	$\{\mathsf{true}, \mathsf{false}\}$
	History	$\mathbb{N} \times$ EXPR	$\mathbb{N}$
	Publish-subscribe	EXPR	*none*

at which the requested slice of data begins, and returns a timestamped list of data values.

The interfaces reveal that the oracle architectures follow a strict separation of responsibilities when it comes to evaluating the conditions attached to performance constraints: Data is requested, and the evaluation itself happens within the smart contract DApp. This pattern is realistic in that it protects the exact operational logic specified in the smart contract—conditions themselves may be confidential and stored in protected areas of the blockchain, or may be implemented using homomorphic encryption or zkSNARKs [32]. Yet, it requires potentially large amounts of data to be transferred.

We thus propose *conditional variants* of the above oracle architectures to reduce the number of transactions and the size of the transaction payload itself. The idea is to externalize the evaluation of a condition from within the smart contract DApp to the oracle, removing the need to transfer any arbitrarily large value of an external data source. The lower half of Tab. 6.2 shows the changed oracle interfaces of the conditional variants, notably including a new parameter specifying an expression from a generic set EXPR of all possible expressions. We do not specify the structure of those expressions in detail, but require them to yield a Boolean result and only reference the value of the external data source the specific oracle is associated with.

The result of querying a conditional oracle variant then reflects the result of evaluating the given expression, e.g., true or false in the case of the conditional storage and request-response oracles. For conditional history oracles, the equivalent of the enablement time is returned, that is, the first timestamp at which the expression evaluated to true starting at the lower bound provided as a parameter. Note that $\top$ may be returned to express that the expression never evaluated to true. Lastly, the conditional publish-subscribe oracle returns no specific value at all. Instead, a transaction is only sent once the expression evaluates to true, i.e., the corresponding event occurs, essentially serving as a signal. Thus, the conditional oracle variants may significantly reduce both the amount of data transmitted and the number of transactions.

6.4 Oracle Implementation and Usage

The feasibility of the event detection approaches and oracle architectures depends on several factors. We will concentrate on one which is particularly relevant in the scope of oracles: the overall cost. To this end, we built a custom prototype based on the Ethereum blockchain and performed several simulations, which are used to evaluate the proposals in realistic scenarios. The prototype and all associated data is available online[1].

[1] `https://github.com/bptlab/blockchain-deferred-choice`

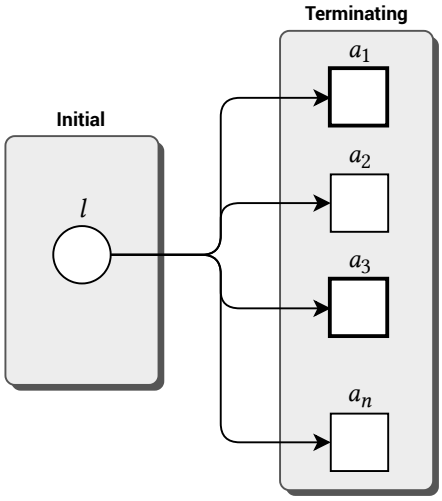


Figure 6.7: Structure of the smart contracts considered in the evaluation

For the implementation, we concentrate on a well-defined subset of the problem space. That is, we consider only smart contracts in which a set of arbitrary actions $\{a_1, a_2, ..., a_n\}$ become active or enabled at initialization, and then engage in a "race" to be performed first (see Fig. 6.7). Only one action can be performed, since they all consume the same legal relation. In the train ticket example, for instance, l leads to the autonomous actions a_1, a_2, a_4 becoming active and the non-autonomous action a_5 becoming enabled at the same time. As discussed in the introduction to this chapter, this situation essentially corresponds to the deferred choice workflow pattern [96], since each autonomous action is attached to an external event whose occurrence eventually enables it. Due to these parallels, some terminology used in the prototype alludes to events and deferred choice.

6.4.1 Overview

Figure 6.8 shows the architecture of the prototype as a UML class diagram. The non-standard DApp and event stereotypes are used for DApp classes and event types, respectively. There are three distinct groups of components, namely (i) those belonging to the simulation framework, (ii) the off-chain oracle providers, and (iii) the on-chain DApps.

Since the prototype is used for comparing the different oracle architectures and event detection approaches, the primary design goal was to provide a level baseline for all of them. No approach should be individually optimized to a higher or lower degree than the others, and they should use interfaces with a similar computational overhead. Another major design decision concerns the distinction between the synchronous and asynchronous oracle patterns, respectively. This results in a visible duality in the class layout, since the oracle providers, oracle DApps, and smart contract DApps are available in both forms. For each oracle architecture, for example the synchronous on-chain history oracle, the prototype contains a set

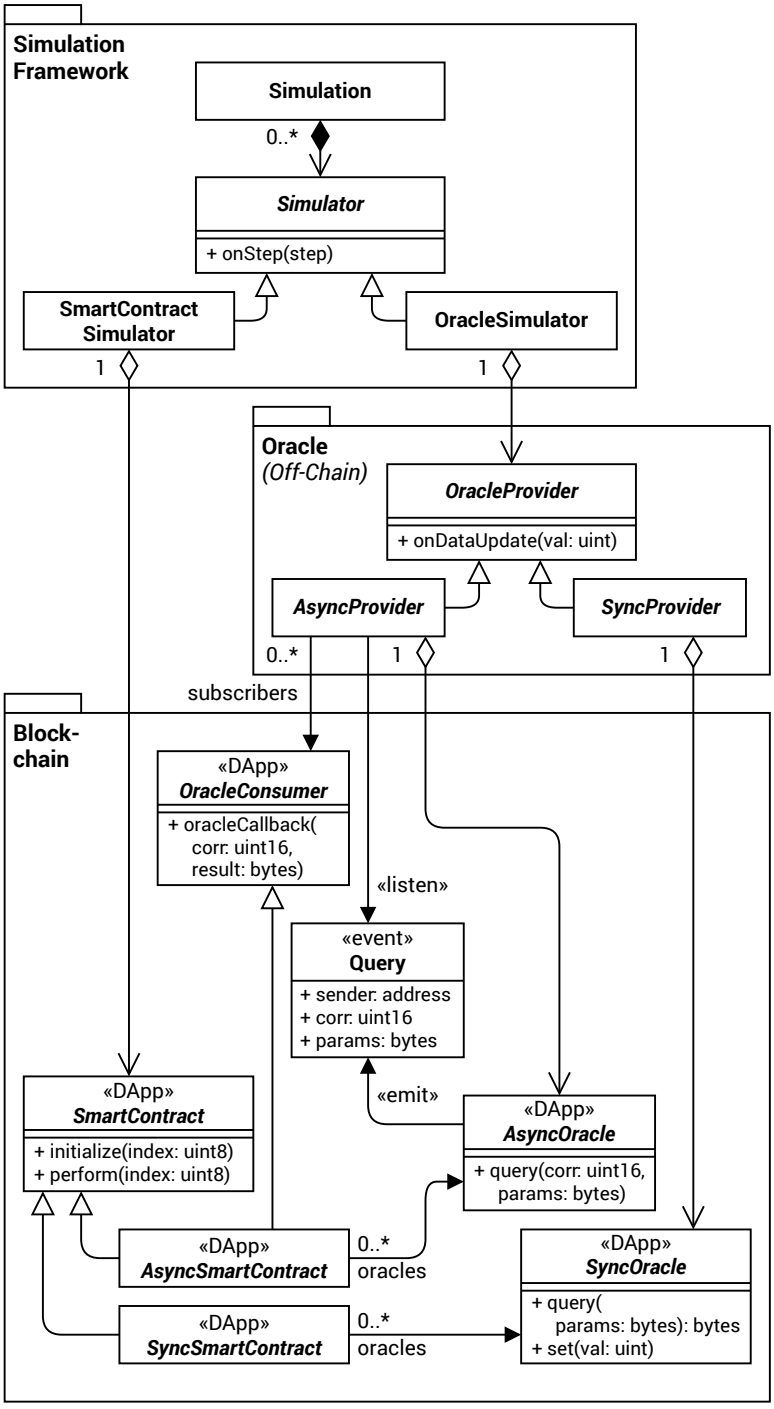


Figure 6.8: Architecture of the prototype developed for the experimental evaluation

of three corresponding concrete classes inheriting from the fitting abstract ones shown in the diagram. In the following, we will walk through each component of the prototype.

6.4.2 On-Chain Components

We use Ethereum [114] as our target blockchain, since it provides all features necessary and its development ecosystem is well-maintained and accessible. The core smart contract logic—reduced to the resolution of deferred choices between autonomous and non-autonomous actions—is contained within DApps, which are implemented using the associated Solidity programming language. For the data domain $\mathbb{D}$, we use uint256, which is the largest static and atomic data type that Solidity allows for. Timestamps are also stored using uint256, and the future timestamp $\top$ is set to the type's largest value $2^{256} - 1$, which is "sufficiently" (see Def. 32) far—several hundred billion years—in the future.

Oracles

The DApp classes AsyncOracle and SyncOracle contain the interfaces of the oracles (see Fig. 6.8). Parameters and query results are encoded in raw byte arrays (bytes) via the same mechanism Ethereum uses to encode transaction payloads. This allows the use of a common interface for all oracles, from which data can be extracted according to specific interfaces (see Tab. 6.2). Larger payloads incur a higher cost following the rules in the Ethereum standard [114].

For *synchronous* oracles, the value or historical values are stored on the blockchain and updated or appended via set. Synchronous oracles may thus directly return a result upon query being called. For *asynchronous* oracles, the off-chain oracle provider stores all data—no setter functions or on-chain storage are needed. Instead, queries are emitted using a custom Query event type containing all the required information for the off-chain oracle provider. Consumers need to extend an additional OracleConsumer class to receive the later callback transaction.

A primary concern for asynchronous oracles is achieving correlation: A consumer needs to be able to link the transaction providing the query result to the query itself. In practice, there are various strategies. For example, some providers like Provable return a unique query ID, which is attached alongside the query result for later matching[2]. In our prototype, consumers choose an ID themselves in the form of the corr value when querying or subscribing to an oracle.

Smart Contracts

The DApp class SmartContract and its children implement the subset of the smart contract semantics discussed above, namely the state and be-

[2] `https://docs.provable.xyz/#ethereum-quick-start-the-query-id`

havior of a deferred choice. Calling initialize via a transaction "starts" the smart contract by setting it to an initial state, which activates all autonomous actions and enables all non-autonomous actions. Calling perform is used to perform an action according to one of the algorithms introduced in Sect. 6.2, potentially resulting in a state transition of the smart contract which picks one of the actions as a "winner" of the race.

In this context, the non-determinism of the smart contract semantics is an issue—if multiple autonomous actions are enabled at the same time, all are valid winners (see Sect. 3.2). DApps are deterministic, though, and one action needs to be chosen. To this end, we opted for a two-phase strategy: Transactions may include a preferred action a_i for each action which is chosen above all others if it is a valid winner. Otherwise, the first valid winner is chosen in the order of the actions' internal indices.

6.4.3 Off-Chain Components

The off-chain components are implemented in JavaScript using Node.js and the Ethereum connector library web3.js.

Oracle Providers

The oracle providers are responsible for bridging the gap between the oracle DApp and the external data source they observe. They manage communication responsibilities, mainly updating the oracle DApp and sending responses to consumer requests depending on the oracle type. In the scope of this thesis, we do not further investigate the connection to the original source of the data.

Simulation Framework

The simulation framework allows the simulation of oracles and a subset of smart contracts in a reproducible way. For example, one such simulation may re-enact the scenario explained at the beginning of this chapter in Fig. 6.2. To this end, a set of Simulator instances replay pre-defined lists of steps on their targets, e.g., sending data updates to an oracle provider or calling the initialization or performance logic on a smart contract DApp.

6.5 Simulation Results

We used the implementation described in the previous section to evaluate the different event detection approaches from two perspectives—the correctness as well as the overall cost. To this end, we designed and simulated several scenarios on a private Ethereum blockchain network with a single node running an official implementation of the Ethereum protocol (Go Ethereum, v1.9.21), using a virtual machine with 12 GB of RAM and

4 CPUs. The detailed specification and raw results of all simulations are available online alongside the prototype.

6.5.1 Correctness

To verify the correctness of implementation, we generated smart contracts following the pattern shown in Fig. 6.7, each with n zero-indexed actions $a_0, ..., a_{n-1}$. An associated simulation timeline was chosen specifically such that the autonomous actions are enabled sequentially in order of their indices, i.e., the associated external events occur in that order as well; that is, a deadline is passed for temporal constraints, or a condition on an external data source becomes true for data constraints. At the same time, the non-autonomous actions are being performed by sending a corresponding transaction as indicated by their index as well. As a result, the action a_0 is always the one that must be performed, since it either becomes enabled first (autonomous) or is actively being performed when no autonomous action is enabled yet.

To serve as a baseline, we implemented the regular storage and request-response oracles as well, and integrated them with the retroactive event detection approach. Each of the 10 oracle variants was then used to simulate $k = 60$ random smart contracts, half of which with $n = 5$ actions and the other half with $n = 10$ actions. With a delay of 60 s between subsequent groups of transactions, the experiment took around four days (99:22:10) in total.

The share of simulations which yielded the correct winner a_0 is shown in Tab. 6.3 for each regular and conditional oracle variant. The newly proposed oracle architectures perform without fail, giving evidence that they indeed describe the intended semantics of smart contracts and that the implementation is accurate. As expected, the traditional oracles encounter issues when certain configurations resembling the problematic example in Fig. 6.2 are generated—they only pick the correct winner in around 35% of cases, i.e., those in which the first action randomly turns out be non-autonomous.

Table 6.3: Share of $k = 60$ simulated smart contracts in which the action a_0 was correctly performed

Event detection approach	*Oracle*	*Correctness*	
		Reg.	*Cond.*
Retroactive	Storage	35%	35%
	Request-response	35%	35%
	On-chain history	100%	100%
	Off-chain history	100%	100%
Publish-subscribe	Publish-subscribe	100%	100%

6.5.2 Cost

Cost is a major factor that influences the adoption of any smart contract enforcement approach in practice. On Ethereum, cost is expressed using *gas*, a stable measure that quantifies the computational complexity and storage requirements of a transaction, and directly translates to its cost in cryptocurrency [114]. We compare the gas cost of all approaches using a series of simulations. Again, the traditional oracle patterns storage and request-response are included for comparison purposes.

Simulation Design

All simulation scenarios follow a pattern in which c smart contracts consisting of exactly one initially active action constrained by a data constraint each access a single shared oracle. The oracle receives u data updates. A perform transaction is sent to each smart contract at each fifth data update, of which only the last will lead to the data constraint's satisfaction by design. This recreates a realistic timeline of events for an oracle with multiple consumer contracts.

All simulations were executed sequentially for each oracle variant and for all combinations of $c \in \{5, 10, 20\}$ and $u \in \{1, 10, 20, 30\}$. Independent sets of transactions were spaced 40 s apart. The experiment took a total time of 25:22:33 to finish.

Deployment Cost

Initially, the DApps need to be deployed to the blockchain network, incurring a one-time deployment cost contingent on the code size. Table 6.4 shows the average deployment costs in gas we have observed. For oracles, there are three significant outliers owing to their more complex code: the regular on-chain history oracle contains code to return the correct slice of historical data, and the two synchronous conditional oracles contain code to evaluate conditions. As expected, they are more expensive to deploy.

For the smart contracts, the differences are less pronounced. There is a clear indication, though, that the externalization of evaluation logic to the oracle for the conditional variants reduces the code complexity of the

Table 6.4: Average smart contract deployment cost

Oracle	*Deployment cost* (average, in 10^3 gas)			
	Oracle DApps		*Smart contract DApps*	
	Reg.	*Cond.*	*Reg.*	*Cond.*
Storage	276 /	408 / +48%	1431 / +3%	1406 / +1%
Request-response	281 / +2%	281 / +2%	1502 / +8%	1477 / +7%
On-chain history	467 /+69%	552 /+100%	1520 /+10%	1386 /
Off-chain history	281 / +2%	281 / +2%	1592 /+15%	1448 / +4%
Publish-subscribe	281 / +2%	281 / +2%	1577 /+14%	1490 / +7%

smart contract itself, and that the more powerful approaches like publish-subscribe and history oracles require larger DApps.

Note that it is moot to attach such gas costs to an amount of fiat currency like USD, since the value of cryptocurrencies is notoriously volatile to market fluctuations and the senders of transactions themselves decide on a fee multiplier to incentivize miners to pick up their transaction faster. The deployment costs shown in Tab. 6.4 would translate to a few cents, up to double-digit dollar amounts depending on these variables. It is thus more helpful to compare the approaches relative to each other.

Operating Cost

The operating cost was derived by dividing the total cost minus deployment costs by the number of consumers c, arriving at an average cost per consumer. The results were normalized globally from the minimum (165,407 gas for the storage oracle with $c = 20$, $u = 1$) to the maximum (1,656,007 gas for the publish-subscribe oracle with $u = 30$), producing the overview shown in Fig. 6.9.

Several observations are immediately apparent: All synchronous oracles (Fig. 6.9a–d) become relatively less expensive the more consumers share the cost, as visible by the decline along the c-axis. This is not the case for asynchronous oracles (Fig. 6.9e–j), as their cost linearly scales with the number of consumers.

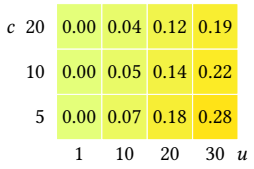
The more updates there are, the more expensive the approaches tend to get, albeit on different scales. This is especially evident for the on-chain history (Fig. 6.9c–d) and the regular off-chain history (Fig. 6.9g) oracles, which show a clear superlinear trajectory on the u-axis because of storage and payload cost increases.

This is not the case for the other oracles, which experience at most a linear growth alongside the number of updates. Notably, while the regular publish-subscribe oracle is the most expensive in our tests, it exhibits a very predictable and linear growth of cost per update, which is not tied to storage or payload requirements.

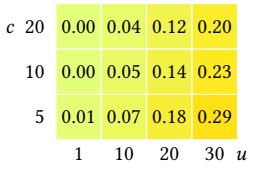
Interestingly, the conditional variants of the storage and request-response oracles are almost exactly as expensive as their regular counterparts. This is mainly due to Ethereum's padding of transaction parameters to words (256 bit), making a Boolean value take up just as much space as an integer. However, for history oracles the effect is very apparent, and for publish-subscribe considerable: the conditional variants outperform the regular variants by a steady margin the larger the payloads get and the more transactions are to be sent.

6.5.3 Overall Feasibility

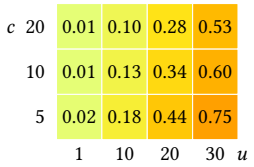
The prototypical implementation and simulation results lend insights into the feasibility as well as the limitations of the event detection approaches



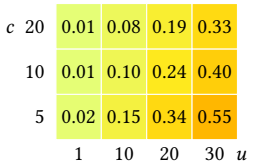
(a) Storage



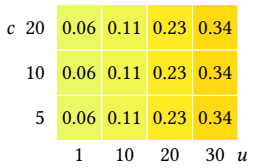
(b) Storage, cond.



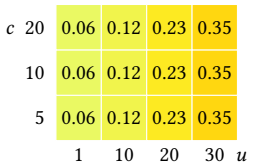
(c) On-chain history



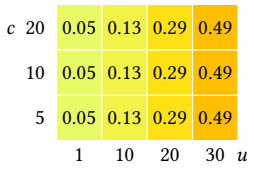
(d) On-chain history, cond.



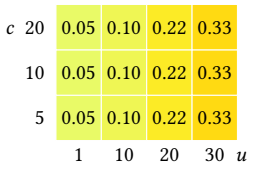
(e) Req/res



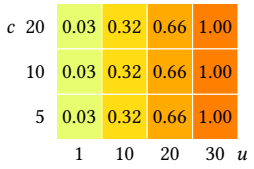
(f) Req/res, cond.



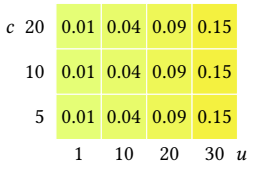
(g) Off-chain history



(h) Off-chain history, cond.



(i) Pub/sub



(j) Pub/sub, cond.

Figure 6.9: Normalized, relative operating cost of an oracle per consumer with the given number of data updates u and consumers c

proposed in this chapter, and thus into the feasibility of the concept of autonomous actions on blockchain networks in general.

Economical Considerations

Considering the results of the simulations, it is evident that only the conditional oracle variants appear to be feasible for environments in which a lot of data is changed frequently. However, the question whether an approach is feasible always also depends on the concrete smart contract. If the external data sources involved infrequently update their value, e.g., once a week, then many of the approaches are economically viable.

The publish-subscribe approach has a major downside, which is that depending on the concrete performance constraints a lot of superfluous transactions are being sent. For example, it is not feasible to send a transaction every time the system time changes for a temporal constraint—since this happens constantly based on the resolution of the time domain, and is naturally bounded by the average block time of the underlying blockchain network. In practice, it might thus be beneficial to separate the performance constraints based on their type, and perform a publish-subscribe approach for data constraints and a retroactive approach for temporal constraints.

Note that we did not consider potential optimizations that would alleviate some of the issues identified above, such as unsubscription mechanisms for the publish-subscribe oracles. Such a feature could be trivially implemented by exposing an unsubscription function in the oracle DApp, which leads to the consumer DApp being removed from the list of subscribers. Similar optimizations could be conceived for other oracle architectures as well, e.g., involving compression or removal of old data.

Transition Scarcity and Backdating

The retroactive event detection approach guarantees that even with delayed or scarce transactions, a performance of autonomous actions in the correct order is guaranteed. With backdating, even the original action timestamps can be reproduced, which is important for relative temporal constraints that may otherwise start too late.

This approach has consequences, though. One is that manual transactions of the participants are still required. There is, by default, no mechanism to force any party to send transactions, or to automatically send transactions to drive the progress of the smart contract. If, for instance, an autonomous action is the last action before a smart contract finally terminates, there is—strictly speaking—no obligation or responsibility for any of the participants to send the transaction that finally executes the action. As such, the approach handles transaction scarcity, but not the absence of transactions overall.

A possible solution would be to artificially add additional transactions, e.g., perform polling [72]. This would introduce the need for another en-

tity submitting these polling transactions in certain intervals. As determined above, though, additional transactions further drive the cost of the approaches, and may quickly end up becoming unfeasible.

Second, while the DApp is certainly able to backdate action performances and reconstruct the correct execution of the smart contract according to the operational semantics, outside services may not participate in this scheme. For instance, a stock option contract which is supposed to autonomously sell stocks once a certain strike price is reached will not be able to get that exact price anymore if the autonomous action is only triggered a week later. In practice, some form of compensation mechanism may need to be implemented in case a later-than-expected performance of an autonomous action leads to problems for a party.

Shifting of Trust

Smart contracts require tamper-proof enforcement via computer code, and the introduction of oracles outside the protected environment of the blockchain network may introduce potential for tampering. That is, every use of an oracle shifts some amount of trust their way. Oracles must be trusted to provide recent data, and promptly react to queries, in order to support the event detection approaches discussed in this chapter.

This is especially true for the conditional oracle variants. While they scale better from a cost-perspective, their externalization of the evaluation of expressions obviously requires even more trust to be put into them. In some cases such expressions may even be confidential and stored in a permissioned blockchain, which would prohibit them from being shared with oracles. However, there are techniques to further secure the usage of oracles from within DApps, some of which could also be applied to our extended oracle architectures [5].

Limitations

The prototype has several limitations. For one, each oracle is used to access exactly one external data source, meaning that there is a 1-to-1 mapping between oracle and data source. In practice, many oracles may provide access to multiple or even arbitrary data sources, e.g., by providing a URL with the query. While such features could be easily added to publish-subscribe oracles, it is not possible to account for this in the general case with history oracles: Naturally, to store historical values of data sources, those must be known before a query arrives.

Second, we did not consider multi-chain environments in the event detection strategies. In case parts of the smart contract are deployed in DApps on different blockchain networks, cross-chain communication protocols would need to be used to synchronize the choice and performance of actions. The added complications certainly deserve more attention in future work on enforcing smart contracts on blockchain networks.

Chapter 7

Time on Blockchain

Deadlines and the notion of time spans or delays are often found in legal contracts and smart contracts alike, and are thus a first-class citizen in our smart contract metamodel (see Sect. 3.1) as temporal constraints. A DApp running as a part of an SCMS must be able to correctly determine whether they are satisfied. After all, an action should not be performed before it is enabled, but neither should the DApp prohibit its execution prematurely.

In the course of this thesis, we have employed a somewhat simplified view on these issues: The assumption was that a transaction tx sent by some participant to execute an action was submitted and then mined with virtually no delay, and that the current system time t of the operating environment state $o = (t, \phi)$ was readily available to the DApp. In practice, however, there may be noticeable delays before a transaction is included after it has been submitted, that is, the transaction timestamp t_{tx} may be different from the system time when tx is mined. Further, DApps only have very limited and peculiar means of telling the system time at all.

In this chapter, we will investigate the issue of telling the time from within DApps (see Sect. 7.1), and compare available methods using various metrics (see Sect. 7.2). The goal is to assess their suitability as to implementing temporal constraints as part of a smart contract enforcement approach (see Sect. 7.3).

Parts of this chapter are based on our previously published work, in which we explored the support for temporal constraints found in business processes within DApps [67].

7.1 Timing of Transactions

Determining the timestamp of a transaction is inherently difficult, but various approximation methods are available.

7.1.1 Technical Restrictions

We consider the smart contract model $\mathcal{M}_{ticket} = (P, D, L, A, C)$ introduced in Sect. 3.2.1 (see page 39) modeling the train ticket smart contract, and further assume it is implemented using a single DApp Ð. Let $a_t := a_5 \in A$ be the non-autonomous action allowing the passenger to cancel their ticket themselves. We extend the example to include a 24 hour deadline for a_t, after which the passenger may not cancel their ticket anymore.

Then, situations as shown in Fig. 7.1 may occur: The passenger decides to cancel the ticket within the allotted timeframe and submits a transaction tx to this effect, but due to delays in the blockchain network the inclusion time d_{tx} is longer than usual and it is not immediately included in a block. As a result, tx is only mined and executed after the 24 hour deadline has expired according to the system time of the operating environment. The state of a_t has, of course, since changed from being enabled to just active and a_t can not be performed.

The passenger may now have grounds to claim that they sent the transaction in time and a_t should have been performed, whereas the railway company may claim that certain delays are to be expected and should be factored in by all participants. While this is ultimately a decision that has to be made by the parties, it comes down to clearly showing intent: The passenger deliberately prepared and submitted a transaction according to the smart contract model, and initiated the necessary steps towards making use of the power granted to them. The same would apply when fulfilling an obligation, or acting on a permission. That is, the underlying technological framework should not influence any contractual agreement.

We will thus assume that the submission of a transaction as signified by its transaction timestamp t_{tx} should be the decisive criterion when determining whether a temporal constraint was upheld. However, the transaction timestamp t_{tx}—as in "the time of submission of tx to the blockchain network"—is usually not available to DApps in blockchain ecosystems such as Ethereum. As a consequence, transactions are mostly timed using the block timestamp t_i of the block $B_i = (t_i, T_i)$ they are eventually in-

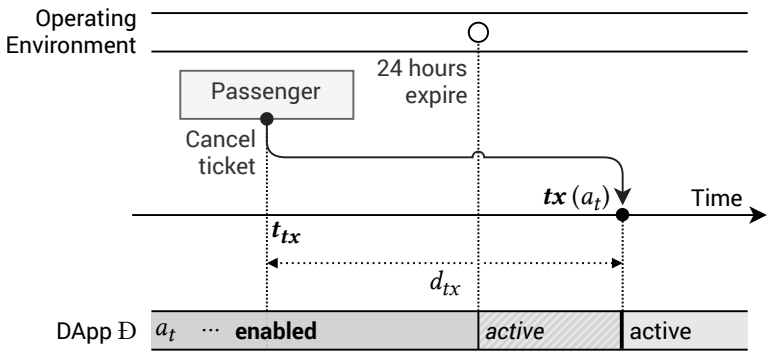


Figure 7.1: Delays in the network leading to an action a_t not being performed

cluded in, since this is the first point in time consensus has been reached. However, there are other methods to approximate the transaction timestamp as well.

7.1.2 Approximation Measures

In the following, we will introduce several *approximation measures*. An approximation measure is a method which allows a DApp Ð to acquire an estimated value for the transaction timestamp t_{tx} of a transaction tx targeting it, either using information already available to Ð or using additional entities and provisions. We will denote an approximation measure using a function-like syntax, i.e., $t_{\mathsf{ANY}}(tx) \approx t_{tx}$, with $t_{\mathsf{ANY}}(tx) \in \mathbb{N}$ and with ANY being replaced by a unique name. For the remainder of this chapter, we will assume a blockchain $\mathcal{B} = (B_0, ..., B_n)$ and a transaction $tx \in T_i$ for some block $B_i = (t_i, T_i)$ of $\mathcal{B}$ is given.

Block Timestamp

The de-facto standard way of determining a timestamp for a transaction is to take the containing block's block timestamp as a reference, which we will define as $t_{\mathsf{BT}}(tx) := t_i$. This is equivalent to the transaction's inclusion timestamp.

Block Number

Most blockchain networks aim at achieving a predictable and regular supply of new blocks. That is, the average block delay, say $\overline{d}$, is supposed to stay reasonably constant over time. Since blocks are numbered consecutively, it is thus possible to deduce an approximation of the block timestamp t_i of a block B_i and in turn the transaction timestamp t_{tx}. We define $t_{\mathsf{BN}}(tx) := t_0 + i \cdot \overline{d}$ where t_0 is the timestamp of the genesis block B_0. In short, the product of the block number and the average block delay are added to the block timestamp of the initial genesis block.

Parameter

Since transactions may carry additional data as a payload, a timestamp can be attached to the transaction locally by the sender of the transaction during its creation and signing. We call that timestamp $t_{\mathsf{PA}}(tx)$. This parameter has to be included specifically in the ABI of the DApp, like in the following listing showing a snippet of a Solidity DApp as used for Ethereum:

Listing 7.1: Solidity DApp function with timestamp parameter

```
function withDeadline(uint timestamp) external {
  require(timestamp < 1608811200,
          "deadline missed");
  // ...
}
```

Notarization

Some blockchain networks and distributed ledgers include so-called notaries, trusted nodes which have some form of normative power in the network, like in the case of Corda [45]. They are trusted to fulfill tasks like transaction validation, signing, or mining depending on the network's configuration. In particular, this notarization process could also include a timestamping service.

That is, on or immediately before the submission of tx to the network's transaction pool, a notary node picks up the transaction and assign a timestamp to it. We call this measure $t_{\mathsf{NO}}(tx)$.

Oracle Measures

Oracles can be used to access arbitrary external data from within DApps. As such, they can of course also be used to query external time servers for the current system time, which can in turn be used for the transaction timestamp t_{tx}. We differentiate between two variants of this technique. One uses storage oracles to get the latest valid timestamp known to the oracle DApp which is already on the blockchain. We call this measure $t_{\mathsf{SO}}(tx)$. Alternatively, request-response oracles can be used to asynchronously query the time of a time server, and check temporal constraints within the callback transaction. We call this measure $t_{\mathsf{RO}}(tx)$.

We do not consider any of the novel oracle architectures which we have introduced in the previous chapter. They were designed with event detection in mind, and do not add significant capabilities that would make them more useful in determining the transaction timestamp than the traditional oracle architectures used above.

7.2 Qualitative Comparison

The approximation measures for the transaction timestamp t_{tx} rely on different mechanisms and exhibit peculiar advantages and drawbacks. To provide a ranking or assessment of these differences, we will compare the measures according to a number of quality metrics. The quality metrics have been selected from metrics often used in evaluating blockchain-based systems (trust, cost, and reliability) or hand-crafted for the timestamp approximation use case (accuracy, retrieval, resolution, monotonicity).

Table 7.1 shows an overview of the results of this comparison, which we will explain in detail in this section. For some metrics, we employ a relative point system, where we rank the approximation measures from best to worst. Note that this does not necessarily imply that the "best" measure is performing well absolutely speaking.

Table 7.1: Relative comparison of the approximation measures, with an assessment in parentheses being subject to exceptions

Measure		*Accuracy*	*Trust*	*Cost*	*Reliability*	*Retrieval*	*Resolution*	*Monotonic*
Block timestamp	t_{BT}	●●○○	●●●○	●●●●	●●●●	Sync.	Block	Yes
Block number	t_{BN}	○○○○	●●●●	●●●●	●●●●	Sync.	Block	Yes
Parameter	t_{PA}	●●●○	○○○○	●●●○	●●●○	Sync.	—	No
Notarization	t_{NO}	●●●●	●●●○	●●●●	●●○○	Sync.	—	(No)
Storage oracle	t_{SO}	●○○○	●●○○	●○○○	●○○○	Sync.	(Block)	Yes
Req/res oracle	t_{RO}	●○○○	●●○○	○○○○	○○○○	Async.	(Block)	No

7.2.1 Accuracy

The deviations of the approximation measures to the transaction timestamp t_{tx} are illustrated in Fig. 7.2, which displays the possible range of outputs for each measure on a timeline.

Due to the assumption that notarization of a transaction happens exactly on or shortly after submission of tx to the transaction pool, t_{NO} is bound to be the most accurate measure. The parameter measure t_{PA} is assigned shortly before, at creation of tx local to the sender. Assuming honest participants—which we will discuss in the section about trust of this comparison—the accuracy of this measure thus depends on how quickly the transaction is sent and delivered to the blockchain network's transaction pool. With modern internet connections and networking technology in place, this gap will likely by significantly smaller than the inclusion delay, making the parameter measure t_{PA} the most accurate after the notary measure t_{NO}.

The block timestamp measure t_{BT} deviates from the transaction timestamp t_{tx} by the transaction's inclusion delay d_{tx}. The actual inclusion delay of a transaction tx varies in practice depending on the configuration of the blockchain network, like an inadequate target block delay resulting in congestion or a backlog of pending transactions in the transaction pool. Furthermore, the maximum fee a sender is willing to pay for the inclusion of their transaction also has a large impact. Overall, these factors have been shown to be predictable in principle [118] and a high accuracy can still be achieved.

The remaining approximation measures are inherently less accurate and one can only gauge them on an interval. The storage oracle measure t_{SO} depends on the frequency of updates by the oracle provider, who stores the current time within the oracle DApp. Since these updates can only be performed in an earlier or the same block as tx, that is B_i, it can be asserted that $t_{\mathsf{SO}}(tx) \leq t_i = t_{\mathsf{BT}}(tx)$. If the oracle provider stops updating the oracle DApp, the measure will become increasingly outdated.

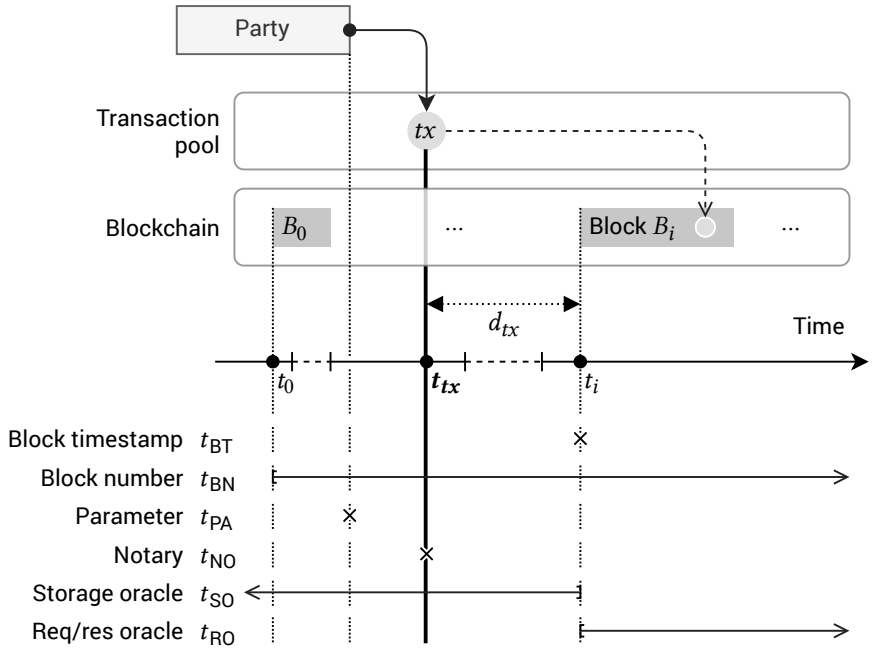


Figure 7.2: Accuracy of the approximation measures

The request-response oracle measure deviates in the other direction. Since the request by the consumer DApp can only be publicly seen after it has been included in a block, this is the earliest the oracle provider can start working on the callback transaction, i.e., $t_{RO}(tx) \geq t_i = t_{BT}(tx)$. If there are delays at the oracle provider or the blockchain network, or the oracle provider ceases to exist, the return transaction may arrive arbitrarily late or even never.

Lastly, the block number measure t_{BT} is the least accurate approximation of the transaction timestamp in general. To illustrate this, we gathered the block times of Ethereum's main network over a span of five years[1]. The data indicates a mean block time $\bar{d}$ of 15.19 s (min. 4.46 s, max. 30.31 s) with a standard deviation of 2.71 s. If these values, including the block timestamp of the genesis block, are inserted in the definition of the measure $t_{BN}(tx)$, this would for example result in a timestamp of 2020-03-29T06:58:49Z for all transactions in block 9690267, when actually it was mined at 2020-03-17T16:55:27Z, some 12 days earlier.

7.2.2 Trust

The overall goal of blockchain technology is the removal of trust from any particular party—external agencies, miners, or participants alike. Likewise, the approximation measures should be as resistant against tampering as possible to support smart contracts.

Figure 7.3 shows a visualization of the parties or blockchain network components trusted for each approximation measure. The block number

[1]Data from 2015-07-30–2020-03-18 via Etherscan

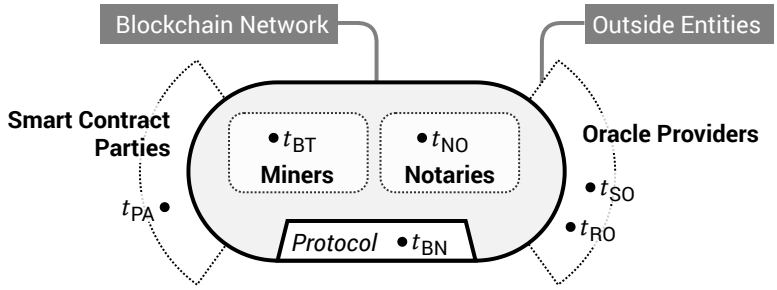


Figure 7.3: Trust of the approximation measures

approach t_{BN} can be considered the least critical since it only relies on the inherent protocol of the underlying blockchain technology, the block numbers. These are guaranteed to strictly increase by one with each block, and there is no party which can in any way tamper with this basic requirement. A miner particularly can not arbitrarily choose a block number, since this would violate the integrity property of the blockchain and be immediately rejected by the network.

This is a difference to the block timestamp measure t_{BT}, which a miner sets when beginning to work on creating a new block. Due to potential network delays and differences in the nodes' local clocks, miners do have some degree of freedom in this aspect: The Ethereum blockchain network, for instance, requires a block timestamp to be "reasonable" [114], whereas the Tezos whitepaper states that the "protocol design must tolerate reasonable clock drift" [38]. To prevent miners from abusing this freedom, Bitcoin even adapted its protocol in 2016 to get rid of a monetary "incentive for miners to lie about the time of their blocks"[2]. However, the actual potential for these kinds of exploits is limited in practice, since consensus algorithms tend to prefer younger blocks and often reject blocks which deviate too much from the current local time of many nodes, e.g., at most 15 seconds into the future for Ethereum[3].

Similarly, the notarization measure t_{NO} puts trust into one or a set of notaries which ultimately pick the timestamp. It is in the best interest of the blockchain network to pick honest notaries, and provide some mechanism to oust dishonest ones, but they still introduce a potential source of distrust. In general, we still assume that miners and notaries are unlikely to have a stake in smart contracts, and can be trusted as an integral part of the blockchain network.

For the oracle measures t_{SO} and t_{RO}, the current system time is acquired through an external oracle provider. The oracle provider is trusted to serve correct information, which is not specific to this particular use case. In fact, oracle providers obviously have an interest in being trusted in general, and some offer various options to mitigate any amount of distrust. Provable, for instance, uses a system that provides proofs to certify that they did

[2]Bitcoin Improvement Proposal (BIP) 113, `https://git.io/JfYqt`
[3]Ethereum Ethash protocol (Go version), `https://git.io/JvdNc`

not tamper with any data received from external APIs before sending it to the consumer DApp[4]. Other methods include second-layer consensus algorithms to combine the output of multiple oracle providers, reducing the influence of each individual entity [74]. In light of the business model and the desired customer retention, we assess the needed trust be only slightly higher than for the network measures.

The parameter t_{PA} performs worst from a trust perspective. While we assume that the parties to a smart contract trust each other enough to enter into a legal agreement in the first place, the purpose of blockchain-based enforcement is precisely to get rid of any potential for tampering. Trusting any other party to attach a correct timestamp to a transaction even if this has detrimental consequences for them is a steep assumption that can not easily be justified in the context of smart contracts.

7.2.3 Cost

The cost of a measure is made up of two components, (i) the enclosing *transaction cost* automatically paid to the network, which is determined by the size of the transaction payload and the computational complexity of the executed DApp functions, as well as (ii) any *oracle fee* payable to the external oracle providers via cryptocurrency transfers. Naturally, each approximation measure is used in a transaction to begin with, which incurs a baseline transaction cost. We derive our ranking in Tab. 7.1 from whether additional costs are created by a measure.

The block timestamp t_{BT} and the block number t_{BN} measures—apart from a trivial and inexpensive calculation for the latter—rely on readily available block information that is "free" to the DApp with little to no additional overhead. Similarly, if a blockchain network provides notarization, this service will most likely be mandatory like in the example of Corda and factored in into any transaction cost.

The parameter approach t_{PA} requires adding additional payload to the transaction, which increases the transaction cost by a small degree. Since timestamps are usually not very big, this cost will most likely be minimal, especially compared to the oracle measures: They require a relatively expensive inter-blockchain function call to the oracle DApp, and are subject to fees imposed by the oracle providers. Provable prices standard API calls at USD 0.01, up to USD 0.04 if a notary proof is requested[5], not even factoring in the callback transaction yet, making the oracle measures by far the most expensive.

7.2.4 Reliability

Reliability is a software quality metric describing whether a system is available when required and non-susceptible to system faults [52]. It can

[4] `https://docs.provable.xyz/#security-deep-dive`
[5] `https://docs.provable.xyz/#pricing`

be applied to the approximation measures as well, since it is critical that they can be obtained at all times by a DApp. For the block timestamp t_{BT} and block number t_{BN} approaches this is inevitably the case as they are part of the blockchain network's inherent protocols. Similarly, the parameter measure t_{PA} does additionally depend on the parties' local environment correctly functioning, but ultimately the parameter is always available if a transaction is being mined.

The remaining measures add considerable uncertainty regarding the reliability, though. For the notary measure t_{NO} the notary nodes present a possible point of failure, especially when they are organized in a second-layer consensus network. The oracle measures even contain a third-party external oracle provider which is not otherwise involved in the blockchain network and unpredictable from a reliability perspective [74]. That means if the oracle provider fails or ceases to be maintained, e.g., for monetary reasons, the measure can not be used anymore. While the storage oracle approach t_{SO} is at least guaranteed to return *some* value in this case—albeit potentially outdated or uninitialized—, the request-response oracle approach t_{RO} could lead to deadlocks in consumer DApps waiting for a callback transaction which is never sent. This is a particular problem for long-running smart contracts since oracle providers which were initially available might cease operations at some point.

7.2.5 Retrieval

Within the transaction tx, the approximation measure needs to be calculated or otherwise retrieved. This happens in a synchronous way for most of the measures, that is, they can be directly retrieved and are immediately usable. For example, the block number and block timestamp are available to the DApp, as well as any parameters or notarized information. While a storage oracle needs to be queried, it likewise immediately returns a usable value since information is already on the blockchain.

Only the request-response oracle measure t_{RO} uses an asynchronous retrieval mechanism. That means that a request needs to be emitted by the DApp before the transaction tx ends, and only in a later separate callback transaction tx' will the approximation be available. This introduces a delay and a high degree of complexity in the implementation.

7.2.6 Resolution

A block $B = (t, T)$ contains not only one, but an ordered list of n transactions $T = (tx_1, ..., tx_n)$. Some approximation measures will exhibit a certain batching behavior, in which all of the transaction in a block are assigned the same timestamp, i.e.,

$$\forall tx_i, tx_j \in T \; : \; t_{\mathsf{ANY}}(tx_i) = t_{\mathsf{ANY}}(tx_j)$$

for some measure t_{ANY}. We say that these measures have a block-level resolution.

This is the case for the block timestamp measure t_{BT} and the block number measure t_{BN}, since they are exclusively calculated from properties of the block B. The oracle measures also exhibit batching, albeit in a slightly different form. For t_{SO}, all transactions between two update transactions to the oracle DApp will receive the same timestamp. This could potentially even span multiple blocks. For t_{RO}, it depends on the implementation of the side of the oracle provider. Assuming an instant processing of all requests and a prompt sending of the response transaction, batching will be visible for transactions within the same block.

Only the parameter and notarization approach work on a transaction-level resolution, and batching does not occur. In a sense, the resolution of the measures is finer, and allows for a comparison between transactions even within the same block.

7.2.7 Monotonicity

As a last metric, we considered whether the measures are monotonically increasing from the perspective of the DApp. That is, when a DApp encounters a transaction tx, it should never happen that the approximation measure returns an earlier timestamp for a later transaction tx'—or, more formally, the following equation should hold for all transactions tx, tx' and for all time measures t_{ANY}:

$$t_{tx} \leq t_{tx'} \implies t_{\mathsf{ANY}}(tx) \leq t_{\mathsf{ANY}}(tx')$$

The block measures t_{BT} and t_{BN} guarantee this property, as does the storage oracle measure t_{SO}: They either stay the same or increase for subsequently mined transactions. However, in particular the transaction-level measures are vulnerable to "overtaking". That is, a transaction tx is submitted before tx', but tx' ends up being mined and included earlier. In case of the parameter and notarization approach, for instance, this could happen due to a higher fee attached to tx' incentivizing miners. The request-response oracle measure t_{RO} is also vulnerable to this, since the callback transactions may overtake each other.

Note that there are different ways of notarizing transactions, and some blockchain networks or distributed ledgers may include the equivalent of mining a transaction in the notarization procedure. In that case, the timestamps assigned would also be monotonic.

7.3 Application to Smart Contracts

The transaction timestamp t_{tx} is essential when evaluating whether temporal constraints are satisfied, and thus certain actions may be performed within tx. There are, in general, two categories of such temporal constraints: those which are contingent on an absolute point in time like a deadline, and those which are relative to previous actions like delays.

Table 7.2: Relative comparison of the approximation measures for implementing temporal constraints

Temporal constraint		*Absolute*	*Relative*
Block timestamp	t_{BT}	●●○○	●●○○
Block number	t_{BN}	○○○○	●○○○
Notarization	t_{NO}	●●●●	●●●●
Parameter	t_{PA}	●●●○	●●●○
Storage oracle	t_{SO}	●○○○	○○○○
Request-response oracle	t_{RO}	●○○○	○○○○

In this section, we discuss whether the approximation measures are suitable for either or even both of these categories of constraints. For this, we consider implementation aspects as well as the quality metrics introduced above to arrive at a relative comparison of the approximation measures. The results are shown in Tab. 7.2.

7.3.1 Absolute Temporal Constraints

Consider a generic absolute temporal constraint c which is satisfied as soon as a deadline $t_\delta \in \mathbb{N}$ is passed. In formal terms, given a smart contract state $s = (\Lambda, v, \mathrm{H})$ and an operating environment state $o = (t, \phi)$, the satisfaction function is defined as follows:

$$\gamma(c, s, o) = \mathsf{true} \iff t_\delta \leq t$$

Figure 7.4 shows a timeline of this scenario. An action a, which has been activated at some point before in the smart contract execution, is enabled when the deadline t_δ is passed and the associated temporal constraint c becomes satisfied. In a transaction tx intended to perform a, the DApp Ð needs to enforce the temporal performance constraint and check whether t_δ has actually passed by comparing it to its transaction timestamp t_{tx}, i.e., evaluate the condition $t_\delta \leq t_{tx}$.

In the extended train ticket smart contract $\mathcal{M}_{ticket}$, the action to cancel the ticket when the passenger's discount card expires is an example for this; the deadline is the date of expiry of the discount card, after which the autonomous action canceling the ticket becomes enabled.

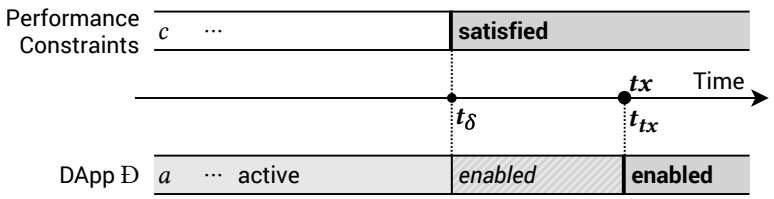


Figure 7.4: Timeline of absolute temporal performance constraints

Absolute temporal constraints are susceptible to false positives and negatives. Let t_{ANY} be any of the approximation measures. A *false positive* occurs when t_{ANY} erroneously reports that t_δ has already passed although it has not, that is, $t_{tx} < t_\delta \leq t_{\mathsf{ANY}}(tx)$. Conversely, a *false negative* occurs when t_{ANY} erroneously reports that t_δ is in the future, although it has already been passed, or $t_{\mathsf{ANY}}(tx) < t_\delta \leq t_{tx}$. Both false positives and negatives may have severe consequences for participants of a smart contract, since actions may be enabled or blocked when they should not be.

Figure 7.5 visualizes whether the approximation measures exhibit false positives or false negatives, and if so, within which intervals. The true transaction timestamp t_{tx} is marked by a bold line. A solid gray background left of this line signifies a false negative if a deadline falls into this interval; a hatched background signifies that a false negative *may* occur depending on the actual deviation. To the right of the bold line the same applies, just pertaining to false positives.

As an example, consider the deadline $t_{\delta 1}$, which is set between the creation of the transaction tx local to the participant and the submission to the blockchain network at t_{tx}. The parameter measure $t_{\mathsf{PA}}(tx)$ will always yield a false negative for $t_{\delta 1}$, since it is fixed at the creation of the transaction tx. The block number approach $t_{\mathsf{BT}}(tx)$ and the storage oracle $t_{\mathsf{SO}}(tx)$, may yield false negatives depending on their accuracy in that instance. Another example, $t_{\delta 2}$ illustrates false positives. Here, the deadline is set during the time tx spends in the transaction pool, i.e., is submitted but not yet included in a block. The block timestamp measure $t_{\mathsf{BT}}(tx)$ will always

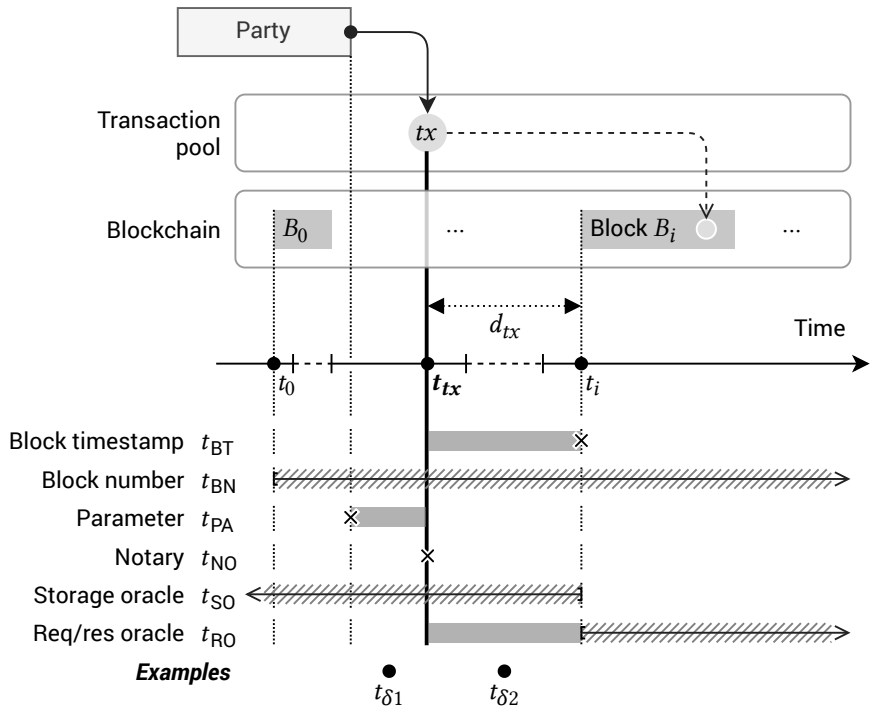


Figure 7.5: Occurrence of false positives and false negatives when detecting absolute deadlines with the approximation measures

yield a false positive here, since the block timestamp is only fixed later during mining. Similarly, the request-response oracle measure $t_{\mathsf{RO}}(tx)$ will also yield a false positive, since the oracle provider will return a timestamp greater or equal to the block timestamp t_i.

In general, the occurrence of false negatives or positives strongly correlates with the accuracy of a measure, which is ultimately reflected in their ranking (see Tab. 7.2). Especially those measures which can only guarantee an accuracy within open-ended intervals will yield far more errors than those which have fixed and short intervals of uncertainty. It should also be noted that Fig. 7.5 is not drawn to scale—the delay between a transaction's local creation and its submission may regularly only account for a few milliseconds, whereas the inclusion time d_{tx} may range from seconds to minutes depending on the blockchain network [118]. The longer the timespan, the more false positives or negatives potentially occur.

7.3.2 Relative Temporal Constraints

Relative temporal constraints are used to express that a certain delay has to be kept between parts of the smart contract. Implementations thus need to compare the timestamps of two transactions, which somewhat multiplies any inaccuracies already present in the approximation measures.

Figure 7.6 visualizes this issue on a timeline. Two transactions tx and tx' are mined and included after each other in two separate blocks. The transaction delta $d_{tx/tx'} := t_{tx'} - t_{tx}$ then is the interval between the two transaction timestamps.

For example, consider a temporal performance constraint c prescribing that an action a may only be performed after a delay d_δ, e.g., 14 days, has passed. Let $t_{\mathsf{active}}(a)$ be the system time when a was activated within a transaction, in this case tx. Then the satisfaction of c may be defined as follows:

$$\begin{aligned} \gamma(c, s, o) = \mathsf{true} &\iff t - t_{\mathsf{active}}(a) \geq d_\delta \\ &\iff t_{tx'} - t_{tx} \geq d_\delta \\ &\iff d_{tx/tx'} \geq d_\delta \end{aligned}$$

In other words, the satisfaction depends on the timestamps of two transactions: that in which the action a was activated (tx), and that in which a

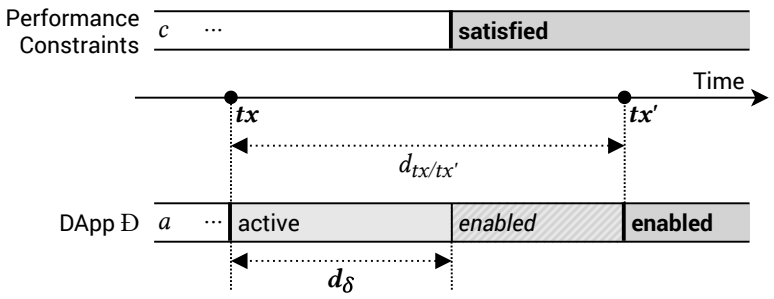


Figure 7.6: Timeline of relative temporal performance constraints

is supposed to be performed (tx'). We assume in the following that transaction tx is included in a block B_i and tx' is included in a block B_j with $i < j$.

Table 7.2 again shows a relative comparison as to the suitability of the approximation measures to support these kinds of constraints. Notably, the block number approach t_{BN} performs slightly better than for absolute constraints since some inaccuracies cancel out, that is:

$$\begin{aligned} d_{tx/tx'} &\approx t_{\mathsf{BN}}(tx') - t_{\mathsf{BN}}(tx) \\ &= (t_0 + j \cdot \overline{d}) - (t_0 + i \cdot \overline{d}) \\ &= (j - i) \cdot \overline{d} \end{aligned}$$

Instead of extrapolating a timestamp all the way from the genesis block, the average block time is only multiplied by the comparatively small difference between the two block numbers. As such, in most cases the influence of extrapolation errors will be smaller.

This is not the case for all the other measures. Especially the oracle approaches may result in arbitrarily bad results, since the open-ended approximation intervals may lead to inaccuracies. Only the notarization and parameter measures benefit from their high accuracy and also perform well when transaction deltas are concerned.

The resolution of an approximation measure further dictates a lower bound for the length of a relative temporal constraints. For example, if the average block time of a blockchain network is 60 s, then a relative temporal constraint demanding that an action is performed within 30 s can not be supported using all approximation measures such as the block timestamp, since it will invariably increase by at least the block time of 60 s. Some measures like the parameter and notarization approach allow for more flexibility even amidst higher block times, though the actual performance of the actions is still bound by the block times.

7.3.3 Usage Guidelines

From our results, it is not possible to conclusively recommend any of the time measures over the others. In practical scenarios, it all depends on the specific and individual requirements: If trust is paramount and some inaccuracy may be tolerable, then block timestamps t_{BT} seem to be a good choice. On the other hand, if the parties do have a certain degree of mutual trust, then the parameter approach t_{PA} is also sensible. DApps may even use more than one approximation measure at the same time, and average or cross-validate them.

For actual implementations, the monotonicity of a time measure is a critical property. Consider, for example, the situation visualized in Fig. 7.7: The passenger tries to cancel their ticket, and sends an associated transaction tx to this effect. The transaction stays pending for a long time, however, and in the meantime their discount card expires. In some other transaction tx', the DApp registers the expiry and autonomously performs a_c,

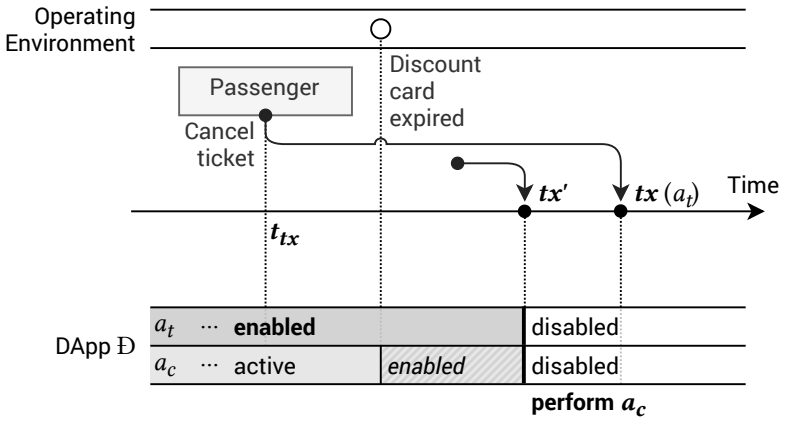


Figure 7.7: A transaction tx' overtaking a previously sent transaction tx

which cancels the ticket without issuing a voucher or refund. Only later tx is mined, at which point a_t is not enabled anymore and the transaction must be rejected.

If the time measure used is monotonic, for example t_{BT}, then it holds that $t_{\mathsf{BT}}(tx') < t_{\mathsf{BT}}(tx)$, and the DApp has no way of figuring out that a_t should have actually had precedence over a_c. Non-monotonic time measures like the parameter or notary approach would allow for this, however, and enable the DApp to roll back or somehow compensate for the erroneously performed action a_t.

Obviously, this introduces a certain degree of uncertainty and implementation overhead, since the DApp would always have to expect the arrival of transactions which invalidate some of the previous smart contract execution. In addition, since the inclusion time of transactions can be influenced by paying higher transactions fees, parties may even try to influence the smart contract execution by strategically trying to overtake other transactions. There is no clear-cut solution to this issue, and parties must agree beforehand on which amount of risk they are willing to take.

Further, it is worthy to note that time is by no means an entirely uniform concept. There are national and international organizations regulating the definition and use of time in certain regions. Occasionally, major or minor adjustments may be made, like changing a time zone for a country. A more subtle example are leap seconds, which are added to UTC in varying intervals to adapt to the slowing of the Earth's rotation. Leap seconds are not reflected in Unix timestamps as used by Ethereum for block timestamps [114], though, and can thus not be accounted for. If a scenario requires leap seconds to be observed, the parameter, notary, or oracle approaches would provide more flexibility after deployment. Corda, for instance, prescribes their notaries to use the "GPS/NaviStar time as defined by the atomic clocks at the US Naval Observatory"[6] for such reasons.

[6] `https://docs.corda.net/docs/corda-os/4.7/key-concepts-time-windows.html`

7.3.4 Limitations and Outlook

Our time measures are built on top of the formal model of blockchains and blockchain networks we introduced in Sect. 2.2. As such, they abstract from some properties and low-level observations. In particular, we do not specifically consider forks, that is, alternative blockchains which might exist at certain points in time until participants find a consensus on which one is the normative version. However, from the perspective of a DApp, forks are invisible to begin with. Rather, forks influence how quickly parties should act upon the change of a smart contract state in a DApp, and reflects the personal risk tolerance.

Some blockchain networks also use other structures and protocols. Hyperledger Fabric [9], for instance, does not use block timestamps at all. Instead, transaction timestamps are assigned by the sender of the transaction themselves, i.e., equivalent to the parameter approach t_{PA} and subject to the same restrictions. *Time-windows* in Corda, on the other hand, are handled by the notary nodes who only accept to commit a transaction when it is inside the window according to the current system time.

Additionally, there are services like Ethereum Alarm Clock[7] which allow the scheduling of transactions at a fixed point in time in the future. In this case, the approach works similar to the request-response oracle, but there is no single off-chain oracle provider. Instead, a second-layer network of "TimeNodes" compete for eventually sending the transaction at the right time for a monetary bounty. Such approaches could also be used for absolute temporal constraints, and would in particular help to remedy the overall absence of transactions (see Chapter 6).

[7] `https://www.ethereum-alarm-clock.com/`

Chapter 8

Related Work

Since this thesis covers large parts of the lifecycle of smart contracts—from modeling and semantics, through system architectures, up to detailed enforceability aspects—a wide array of existing work is relatable.

We identified three broad categories of related work (see Fig. 8.1): First, there are approaches rooted in the legal domain, which have a strong connection to legal contracts (see Sect. 8.1). Second, there are approaches which originate in the BPM domain, mainly concerned with blockchain-based process execution (see Sect. 8.2). Third, there are generic approaches which are not attributable to a specific domain, but still relate to the work presented in this thesis (see Sect. 8.3).

8.1 Legal Focus

In the legal domain, one of the major questions being discussed is whether smart contracts—both as an isolated concept and in connection to DApps—can actually be viewed as a substitute to legal contracts. This pertains to both the requirements on initially forming a legal contract, i.e., offer, acceptance, and consideration [37], as well as on the legitimacy and possible extent of automatic tamper-proof enforcement [111, 103].

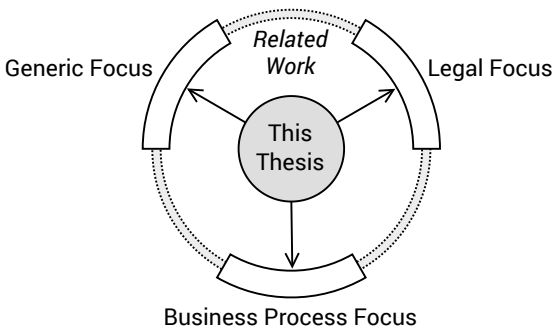


Figure 8.1: Categorization of related work

Despite this uncertainty regarding the eventual standing of smart contracts in the legal domain, which is out of scope of this thesis, numerous research frameworks are being proposed and start-up companies are being established. Being mainly based on blockchain networks, there are parallels with the SCMS architecture and enforcement approach introduced in this thesis on several layers, which we will contrast in the following.

Research Frameworks

Hazard and Haapio introduce the concept of *wise contracts*, which are based on the Ricardian triple smart contract structure (see Sect. 2.1.3) [44]. They use the *prose object* data model that has been introduced by the *CommonAccord* project[1] to represent the non-operational legal prose aspect of the smart contract. Since a prose object is a machine-readable codification of contract terms, it can directly reference and be referenced from the associated operational code. This essentially solves the interlinking issue found in Ricardian triples. The authors provide many valuable insights into how such wise contracts could be safely transmitted and signed, but stop short of tackling low-level automatic enforcement issues as we did in this thesis.

Notland et al. introduce the *minimum hybrid contract*, a construct immutably linking on-chain DApps and off-chain contract documents [86]. Like in the wise contract approach, their focus is on the contract lifecycle stages immediately preceding the performance, i.e., the signing, notarization, and storage of the minimum hybrid contract. The capabilities of the underlying blockchain technology are employed for financial transfers and auditability, but not specifically for enforcement in the sense of this thesis. Temporal constraints, autonomous actions, and related concepts are not mentioned.

Tsai et al. created the *Beagle* framework, which comprises a five-step approach to smart contract development: a preliminary domain analysis, followed by a template-based formal model creation, verification and validation, execution, and runtime monitoring [107]. The authors only give a brief overview of the whole method, though, and only superficially touch on the issues of external data sources and monitoring as well as temporal constraints. There are no provisions for multi-chain enforcement.

Tateishi et al. propose a template-based approach of modeling and enforcing smart contracts [106]. They start with a contract document that is first transformed into an instance of a textual smart contract expression language [97], then into a state chart, and eventually into DApp code for Hyperledger Fabric. Again, while the authors acknowledge the importance of temporal constraints, they are not discussed and a code example suggests support is preliminary, e.g., the current day in a stock option example is modeled as an internal integer variable. Oracles or multi-chain scenarios are not mentioned.

[1] `http://www.commonaccord.org/`

Azzopardi et al. introduce an interesting issue, which is that enforcing a legal contract using DApps is not done by only allowing permitted and rejecting other behavior [11]. They argue that *attempting* to perform an action which is not permitted or which is prohibited, i.e., sending a transaction to claim a refund even though this is currently not possible, constitutes a violation in itself that should be monitored and potentially sanctioned. The model and tooling we propose in this thesis and none of the other approaches mentioned in this chapter take this into account. Some blockchain networks also log rejected transactions, though, which could then be used by monitoring or evaluation tools attached to the SCMS.

Many research papers are focused on individual aspects of smart contract modeling and enforcement, and do not aim to provide fully featured frameworks. Previous research in this area has often been focused on expressiveness, however, specifying and formalizing legal contracts for use in electronic systems, like in the seminal logic model of Lee [73] or the extension of the μ-calculus by Prisacariu and Schneider [92]. Kabilan and Johannesson describe a multi-tier contract ontology expressed using UML class diagrams which contains interconnected legal relations and actions, working towards combining non-operational and operational contract aspects in the same model [56]. In another work, Kabilan applies their ontology to BPMN models [55]. More recently, Flood and Goodenough used deterministic finite automata with attached natural language legal consequences and correlates to analyze legal contracts from the financial domain [35]. Most approaches do not broach the topic of automatic or tamper-proof enforceability, however, for example lacking a clear semantics which is central to the context of this thesis.

Legal Tech Projects and Start-Ups

While many aspects of the legal domain are already heavily automated and digitized, e.g., regarding the storage and indexing of correspondence, smart contracts still remain largely academic and are only just entering the commercial market. With the advent of blockchain and distributed ledger technology in general, a large community of projects and businesses have evolved in the legal tech domain. Note that the line between a legal smart contract development approach and a generic DApp development approach (see Sect. 8.3) are in some cases a matter of interpretation.

Legalese[2] is a project that is "working on the drafting of legal documents the way programmers develop software". Apart from offering several products for digitally handling specific types of contracts like service contracts for contractors and non-disclosure agreements, the goal is to develop a DSL called *L4* based on the modal μ-calculus capturing the logic and deontics of law. However, the capabilities of the Legalese offerings currently seem to be restricted to document generation solutions, and there

[2] `https://legalese.com/`

does not seem to be a generic enforcement approach using DApps yet, although they are sometimes mentioned.

The *Accord Project*[3], part of the Linux Foundation, aims at developing a common format for smart contract specification which integrates with arbitrary distributed ledgers and blockchain networks. Efforts include a template-system for machine-readable natural language contracts called *Cicero* and a textual DSL called *Ergo* capturing the operational aspects of legal contracts. Together, they essentially resemble the components of the Ricardian triple (see Sect. 2.1.3). It is not clear if and how problems like competing actions or enforcing temporal constraints are solved from the publicly available code, i.e., early prototypes of a possible Corda and Hyperledger Fabric integration.

OpenLaw[4] is built around Ethereum DApps, and provides a custom textual markup language allowing the creation of smart contract models with both natural language text and operational instructions interlinked. Forms are automatically generated from the models to populate parameters, and a secure signing protocol is used to deploy DApps based on the model. Since OpenLaw uses Ethereum, it is subject to the isolation and non-continuity properties. To this end, they include both oracles as well as a "relayer" component, which periodically sends transactions to the generated DApps. Again, there is no discussion of the integrity and trust problems this introduces.

Daml[5] is a textual smart contract language built around an abstract model of so-called *Daml ledgers*. Given a transformation from the Daml ledger to an arbitrary blockchain network, it can be used to execute Daml smart contracts. The abstract Daml ledger uses a fuzzy interpretation of time. Each transaction has a "ledger time", which is a timestamp attached by the submitting participant and thus equal to the submission timestamp and comparable to the t_{PA} approximation measures introduced in this thesis. Additionally, there is the "record time", which is the time the transaction was added to the ledger, i.e., the inclusion or transaction timestamp. To ensure some guarantees, Daml allows a configurable minimum and maximum skew between the two timestamps, and also takes into account a per-ledger average transaction delay.

Lastly, some distributed ledgers themselves are specifically built with the vision of smart contracts in mind. Corda is an example for this, with Brown et al. even stating in the first version of their whitepaper that they "envision a future where legal agreements such as business contracts are recorded and automatically managed without error" [17]. Smart contract templates, i.e., parametrized contract documents attached to DApps, are still an integral part of their architecture [45]. There is no dedicated modeling language and semantics for them, however. Still, Corda provides

[3] `https://accordproject.org/`
[4] `https://www.openlaw.io/`
[5] `https://daml.com/`

features like *schedulable states*[6] allowing DApps to schedule timed future transactions within a Corda node, which could be used to alleviate some of the problems discussed in this thesis.

Overall, we conclude that this thesis provides a novel, holistic view of a SCMS with enforceability using DApps as its core component. We consider aspects like multi-chain environments, temporal constraints, and data constraints including external data monitoring which have largely been neglected previously.

8.2 Business Process Focus

Blockchain technology has been identified as a major source of opportunity in BPM [83]. Especially choreographies and inter-organizational processes between mutually distrustful participants who collaborate towards a common business goal benefit from the auditing, monitoring, and enforcement capabilities provided by DApps [109]. As a consequence, blockchain technology has been widely tested and applied to all phases of the business process lifecycle, with research still ongoing [36, 22].

Since we based our design of an SCMS on that of a BPMS, some of the approaches in blockchain-based business processes are relevant in the context of this thesis. In the following, we give an overview of the most prominent approaches with respects to their architectural decisions regarding the BPMS or process engine and their integration with blockchain technology, relating to our first research question RQ1. We also assess their capabilities when it comes to enforcing the business process equivalents of temporal and data constraints using external data, possibly in a deferred choice setting, relating to our second research question RQ2.

An overview is shown in Tab. 8.1, where a check mark means that a topic is present in the accompanying publication. Note that this does not necessarily mean that an implementation or analysis was provided, as clarified in the following for each approach.

Multi-Chain Approaches

Azzopardi et al. present a business process monitoring approach using multiple blockchains, which is structurally similar to—and postdates [71]—the SCMS approach introduced in this thesis [12]. A business process collaboration, modeled as a BPMN collaboration diagram, is potentially split up into multiple fragments deployed in individual DApps. An off-chain notary is used to allow for cross-chain communication and asset transfers to enact the overall process. The notary can also be used to include completely private process parts, which are not run on any blockchain. The authors identify the need for an off-chain notary to be a weak point of their approach, which might be mitigated by using more elaborate cross-

[6] `https://www.corda.net/blog/scheduling-time-based-events-on-corda/`

Table 8.1: Related work with a business process focus, ordered by whether they discuss certain topics and issues

Approach	*Year*	*Multi-chain*	*Temporal constraints*	*External data constraints*	*Deferred choice*
Azzopardi et al. [12]	2021	Yes	—	✓	✓
Adams et al. [3]	2020	(Yes)	—	—	✓
Falazi et al. [34]	2019	(Yes)	—	—	—
Abid et al. [1]	2020	—	✓	—	—
Klinger and Bodendorf [60]	2020	—	—	✓	✓
López-Pintado et al. [76]	2019	—	—	✓	✓
López-Pintado et al. [75]	2019	—	—	✓	✓
Lu et al. [77]	2020	—	—	✓	—
Weber et al. [109]	2016	—	—	✓	—
Ladleif et al. [70]	2019	—	—	—	✓
Corradini et al. [26]	2020	—	—	—	✓
Madsen et al. [79]	2018	—	—	—	✓
Alves et al. [8]	2020	—	—	—	—
Schinle et al. [98]	2020	—	—	—	—
Sturm et al. [101]	2018	—	—	—	—
Sturm et al. [102]	2020	—	—	—	—

chain communication protocols. Overall, the approach seems promising and is, to the best of our knowledge, the only one to implement working cross-chain communication for multi-chain process enactment.

Adams et al. propose the notion of a *blockchain-integrated BPMS* as a light-weight solution that retains traditional capabilities of BPMSs and only interfaces to blockchain networks for reliable storage of data and executing "key contractual terms" [3]. This especially allows the use of more than one blockchain network in the same process, although their YAWL-based prototype only implements a connection to Hyperledger Fabric. There is no discussion about plans or potential issues regarding cross-chain capabilities. The authors claim that their architecture allows the usage of all workflow patterns, since they are still executed locally in the participants' BPMSs. It is not clear, however, how temporal constraints and deferred choice patterns can then be enforced by a tamper-proof DApp, which is the main focus of this thesis.

Falazi et al. introduce the *BlockME* approach to connect processes and blockchain networks [34]. Essentially, the authors propose an extension of BPMN process models that adds elements targeting mostly technical properties, features, and events of blockchain networks, e.g., submitting or receiving transactions or reacting to forks. The approach particularly allows the connection of a single process instance to multiple blockchain networks. The extension elements are then transformed into standard-compliant regular BPMN process models, which can be executed in reg-

ular BPMSs. There is no enforcement of any process logic or constraints, which again is essential in the context of smart contracts.

In summary, we identified only few approaches who support or provision for more than one blockchain network being used for a single process instance, with only one approach by Azzopardi et al. actually implementing some degree of cross-chain enforcement [12].

Temporal Constraints

Some approaches briefly mention challenges involved in enforcing temporal process constraints within DApps [70, 109], or even seem to support them to some degree without disclosing an implementation [60]. Only one approach describes and provides an implementation for temporal constraints: Abid et al. extend Caterpillar to allow for intra and inter-activity temporal constraints like task durations or absolute start/end times [1]. Their implementation is based on Ethereum and uses the block timestamp (see t_{BT} in Chapter 7) to implement guards that reject transactions violating temporal constraints.

Research on temporal aspects in blockchain-based process execution seems to be largely focused on the more technical challenges, e.g., confirmation times, of blockchain networks. Yasaweerasinghelage et al., for instance, evaluate and predict their influence of business process execution [118]. Haarmann uses annotated BPMN choreography diagrams to estimate their total execution time in face of varying inclusion and block delays [42]. None of the approaches mentioned, however, discusses the feasibility or consequences of using one or more time measures.

External Data Sources and Constraints

Caterpillar, one of the most advanced blockchain-based business process engines [76], our own work [70], and many others support internal process data stored in the DApp with associated data constraints, for example using data-based exclusive gateways or conditional intermediate catching events. Since this data is part of the process state and only changes within transactions to the process DApp, however, there are no fundamental issues regarding the non-continuity or isolation properties of the DApps which are the focus of this thesis (see Sect. 6.1)

Some approaches provide support for oracles to acquire external data: Azzopardi et al. state that their cross-chain notary node can also be used as an oracle provider [12]; the trigger components in the seminal approach by Weber et al. can likewise assume the role of an oracle provider [109]; Caterpillar and an associated interpreted approach use service tasks [76, 75]; Lorikeet includes on-chain asset registries [77]; and Klinger and Bodendorf employ black-box message exchanges which may stem from oracles [60]. Almost all approaches provide some form of generic scripting capability, e.g., via script tasks, which can be used to access oracles, but not necessarily react to any callbacks.

The goal of this thesis and in particular Chapter 6 was to additionally allow blockchain-based enforcement of constraints referring to regularly changing external data sources, achieving a kind of continual monitoring of the operating environment state. None of the approaches we assessed discussed these issues.

Deferred Choice

Lastly, deferred choice is a common pattern in business processes [96]. The event detection approaches introduced in Chapter 6 of this thesis are suitable for implementing deferred choice within DApps even in the presence of temporal and data constraints which depend on events occurring outside the control of the blockchain network and outside of transactions.

None of the related approaches supports deferred choice to this degree. For example, Caterpillar allows the usage of event-based gateways—the modeling element used in BPMN to express deferred choice—, but the events following it are all internal and happen during transactions, like errors or signals thrown in other parts of the process [76]. In the choreography-based approach by Corradini et al. the events competing in the deferred choice are all directly mapped to transactions, which leads to the choice being resolved by the ordering of the transactions itself [26]. This is the case for all other approaches as shown in Tab. 8.1, as far as we were able to tell from the publications and open-source implementations.

The insights from this thesis thus provide a considerable step towards supporting complex workflow patterns like deferred choice, temporal and data constraints, as well as multi-chain enactment of business processes in the future, which have until now been largely missing from relevant research.

8.3 Generic Focus

In addition to approaches using DApps and blockchain technology to directly enforce smart contracts or business processes and choreographies, there is a growing interest in developing generic DApps in a secure and structured way:

Due to the immutability of DApps and the complexity and unfamiliarity of their code and design patterns, simple programming errors could have grave consequences. For example, the famous TheDAO bug resulted in the theft of around USD 60 million from a DApp in 2016 [78]. The aim of most model-driven DApp development approaches is thus to allow a secure development of DApps which makes them less susceptible to potential vulnerabilities, according to several recent surveys [51, 7, 100].

Mavridou and Laszka introduce a code generation approach to design DApps for Ethereum using finite state machines [81]. The goal is to reduce the probability of suffering from DApp vulnerabilities by using the strict structure and rigorous semantics of state machines, which abstract

from many of the problematic constructs like external events which we have identified in this thesis. Notably, though, their approach is among the few we could identify which specifically include temporal constraints: Transitions can be annotated with time guards, which use the block timestamp (t_{BT} in Chapter 7) of the actual transaction to check for deadlines or delays. However, there is no critical discussion of the suitability of the block timestamp or potential alternatives. A subsequent extension of the approach with extensive model-checking capabilities exhibits similar restrictions [82].

Andrychowicz et al. use timed automata to model Bitcoin scripts, instructions that can be attached to Bitcoin transactions and which allow for limited DApp functionality [10]. The timed automata can then be verified using a model-checker. Unfortunately, the approach does not include a code generation component, so implementation and model may diverge and potential enforcement issues remain unexplored.

Kasinathan and Cuellar present an interesting approach on DApp modeling for IoT scenarios, e.g., smart manufacturing and building automation, using extended Petri nets [58]. They add a special type of place called "oracle place", which represents input from an oracle. They further briefly describe the semantics of such Petri nets and attached constraints when executed within DApps, subject to the non-continuity property. However, the issues of deferred choice and temporal constraints are not mentioned or solved.

Choudhury et al. use domain-specific ontologies and rules as the basis of their DApp code generation [21]. As an example, they provide an ontology of eligibility criteria of a driver in the context of a car rental. Together with an associated rule describing the conditions of being eligible, code can be generated. The approach, however, only supports static evaluation of data already contained in a DApp or provided via parameters. Similar restrictions apply to the work by Haarmann et al., who generate DApp code from decision tables specified using the Decision Model and Notation (DMN) standard [43].

Oracle Architectures

On a side note, we also introduced several novel oracle patterns in this thesis (see Sect. 6.3). While we used them specifically to monitor external data sources in the context of autonomous actions, they are by no means limited to this use case. To the best of our knowledge, the exact mechanisms of history and publish-subscribe oracles as described in this thesis have not been thoroughly discussed or evaluated in research on blockchain oracles before. Instead, existing research seems to focus on trust issues [46], integration aspects [80], or reliability [74] of the more traditional patterns.

Indeed, in a recent exhaustive literature survey, Al-Breiki et al. identify three major oracle patterns [5]: Two of those correspond to the traditional storage and request-response oracle patterns—where the former is called "immediate-read". The third pattern is named publish-subscribe pattern,

but is fundamentally different to our proposal of the same name: An off-chain or on-chain flag is maintained by the oracle provider and can be manually queried by consumers DApps, making it unsuitable for implementing our event detection approaches. This further indicates that our extension of the oracle architectures provides an unexplored direction.

Chapter 9

Conclusion

We opened this thesis with the development of two research questions: first, how a management system for smart contracts built on blockchain technology may be designed (RQ1); and second, whether DApps can be used to correctly and autonomously enforce smart contracts (RQ2). In the course of the thesis, we have introduced, developed, and evaluated possible answers to these questions, which we will outline and discuss in this concluding chapter.

9.1 Contributions

The contributions of this thesis can be roughly divided into two groups following the research questions, and are summarized as follows:

Notion of Smart Contract Management Systems (SCMSs)
: The SCMS approach (see Chapter 4) unites insights related to smart contracts and DApps from our own research, from the legal domain, and from the BPM community in a single software architecture. We gave an overview and justification of the architecture's components in light of practical multi-chain environments, and incorporated aspects which similar proposals are missing. The SCMS approach suggests an answer to RQ1, and provides an end-to-end vision of how smart contracts may be managed, built, and eventually enforced.

  We have evaluated the SCMS architecture using a proof-of-concept implementation dubbed Mantichor (see Chapter 5). While Mantichor does not cover all features, it still generated valuable insights into how a more complete implementation of an SCMS may perform in practice as blockchain technology continues to mature.

Enforcement of smart contracts using blockchain technology
: In addition to their discussion in the context of the SCMS approach, we have considered more specific enforceability aspects of smart

contracts using blockchain technology (RQ2). These pertain to enforcing data and temporal performance constraints, which are not widely covered in previous research in this area.

For data performance constraints, we proposed two strategies as to how external data can be monitored and evaluated from within the confines of the DApp on a blockchain network. The strategies take into account the problem of competing actions, in which the DApp has to ensure that ordering and precedence conflicts are resolved. We evaluated our proposals using a tailor-made prototypical implementation and simulation, which reveals their cost and feasibility (see Chapter 6).

For temporal performance constraints, we gave a structured overview of the peculiar methods of telling the time from within blockchain networks, and systematically compared their advantages and drawbacks (see Chapter 7). We gave pointers as to how this affects the evaluation of temporal performance constraints, and came to the conclusion that there can be no single usage guideline.

While the above contributions directly relate to our initial research questions, we also consider some products or insights of this thesis which were generated along the way to be major additional contributions:

Model and semantics of smart contracts
: In Chapter 3 we introduced a model and semantics of smart contracts. Ultimately a necessary tool to argue about requirements for and enforceability aspects of smart contracts in the context of this thesis, the metamodel in particular is useful beyond this scope. We have shown this utility of the metamodel by assessing the BPMN choreography diagram standard as to its suitability for smart contract modeling. Thus, the model provides a versatile tool for future research on smart contracts, and the evaluation of new modeling approaches.

Web-based modeler for BPMN choreography diagrams
: Originally developed solely for its eventual use in Mantichor, our web-based modeler for BPMN choreography diagram, chor-js (see Sect. 5.4), has grown to be an independent and recognizable tool in the wider BPMN community. With chor-js, we facilitate the use of choreography diagrams, which did not yet achieve the same degree of popularity as their process or collaboration diagram counterparts since their introduction in the second version of the standard [87]. The tool is available online and continues to be maintained.

Novel oracle architectures
: The history and publish-subscribe oracle architectures introduced in Sect. 6.3 specifically serve to support our two approaches at evaluating data performance constraints, but are usable in other scenarios

as well. Many applications implemented using DApps face similar issues, and may find these capabilities to be useful.

Transaction timestamp approximation measures
Lastly, our structured definition and assessment of transaction timestamp approximation measures in Sect. 7.1 is not only helpful in smart contract or process settings, but for DApp development as a whole. The insights and methods described are easily transferred, and are of potential interest to the wider blockchain community in general.

Taken as a whole, we consider our contributions to consistently advance the state of the art in smart contract management and enforcement. We have provided novel insights, strategies, and results which may provide a foundation allowing the eventual widespread use and acceptance of autonomous smart contracts.

9.2 Limitations and Future Work

Naturally, there are limitations to our work, many of which we have already discussed in the parts of the thesis they emerged in. To conclude, we want to briefly revisit the major limitations, discuss their impact, and state how they may motivate future work in the area.

Perhaps most notably, we placed a focus on relatively traditional blockchain data structures as pioneered by Bitcoin [85] and Ethereum [114]. Our formal model of blockchain networks (see Sect. 2.2) thus may not apply to all blockchain networks or distributed ledgers in general, which in extension limits the applicability of the results founded on the model. While this was a necessary decision to keep the scope of this thesis in check, some distributed ledgers may neither be susceptible to the issues nor the solutions described in this thesis. For instance, this could be an explanation for the difficulty we had to adapt our approach to Corda (see Sect. 5.3), since Corda notably does not use a block structure.

Yet, many questions still remain open even for the commonly used traditional blockchain networks which we covered in this thesis: While our SCMS approach is fundamentally based on the assumption that multi-chain environments will be the norm in practice, we did not consider or implement any concrete cross-chain communication techniques in our SCMS architecture and Mantichor proof-of-concept implementation. As the required protocols and platforms grow more mature, a more thorough investigation into the practical feasibility of the multi-chain assumption is needed. This may include another effort to implement a SCMS, which supports more features and also implements as well as evaluates detailed enforceability aspects which were discussed in a more formal setting in this thesis.

In a similar vein, there is a noticeable lack of modeling support for smart contracts, even taking into account our targeted metamodel. Privacy

and security parameters must be configured in the smart contract model to decide on a suitable distribution across blockchain networks, for which early works including ours from the BPM domain may provide some inspiration [70, 63]. Optimally, these parameters would be susceptible to formal analysis to determine whether a given model fulfills use case specific requirements, and can be safely used without consistency or disclosure issues.

This also concerns further properties and components of legal contracts, for example exchanging goods and money. Blockchain networks are especially suitable for financial transactions considering their deep integration with cryptocurrencies, and novel developments like Non-Fungible Tokens (NFTs) promise to extend these capabilities even further to arbitrary off-chain and on-chain assets. These features should be accounted for in smart contract specification and enforcement approaches, e.g., by supporting payments, escrow schemes, and autonomous asset transfers. The DApps enforcing the smart contract could hold and manage these balances automatically, eliminating the need for middlemen and third-party providers. Common contracts like stock options or rental contracts, which mainly rely on the transfer of money, could as a consequence benefit substantially.

Encapsulating these conceptual issues is the need for a commonly understandable notation for smart contract models [24]. It is essential for the acceptance of smart contracts in practice for them to be at least as accessible as their traditional legal contract counterparts. Our work gives some hints to this end, but does not ultimately solve the issues. This and related challenges—for example, whether fully autonomous smart contracts are even desirable given the flexibility granted by the ambiguity of natural language and the legal system—will shape the future of smart contracts.

Bibliography

[1] Amal Abid, Saoussen Cheikhrouhou, and Mohamed Jmaiel. Modelling and executing time-aware processes in trustless blockchain environment. In Slim Kallel et al., editors, *Risks and Security of Internet and Systems, CRiSIS 2019*, volume 12026 of *Lecture Notes in Computer Science*, pages 325–341. Springer, Cham, 2020. doi: https://doi.org/10.1007/978-3-030-41568-6_21.

[2] Greta Adamo, Stefano Borgo, Chiara Di Francescomarino, Chiara Ghidini, and Marco Rospocher. BPMN 2.0 choreography language: Interface or business contract? In Stefano Borgo et al., editors, *Proceedings of the Joint Ontology Workshops 2017*, volume 2050 of *CEUR WS Proceedings*, 2017. URL http://ceur-ws.org/Vol-2050/FOMI_paper_2.pdf.

[3] Michael Adams, Suriadi Suriadi, Akhil Kumar, and Arthur H. M. ter Hofstede. Flexible integration of blockchain with business process automation: A federated architecture. In Nicolas Herbaut and Marcello La Rosa, editors, *Advanced Information Systems Engineering*, volume 386 of *Lecture Notes in Business Information Processing*, pages 1–13. Springer, Cham, 2020. doi: https://doi.org/10.1007/978-3-030-58135-0_1.

[4] John Adler, Ryan Berryhill, Andreas Veneris, Zissis Poulos, Neil Veira, and Anastasia Kastania. Astraea: A decentralized blockchain oracle. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1145–1152. IEEE, 2018. doi: https://doi.org/10.1109/Cybermatics_2018.2018.00207.

[5] Hamda Al-Breiki, Muhammad Habib Ur Rehman, Khaled Salah, and Davor Svetinovic. Trustworthy blockchain oracles: Review, comparison, and open research challenges. *IEEE Access*, 8:85675–85685, 2020. doi: https://doi.org/10.1109/ACCESS.2020.2992698.

[6] Robert Alexy. *A Theory of Constitutional Rights*. Oxford University Press, 2002. ISBN 9780198258216.

[7] Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. On the verification of smart contracts: A systematic review. In Zhixiong Chen et al., editors, *Blockchain – ICBC 2020*, volume 12404 of *Lecture Notes in Computer Science*, pages 94–107. Springer, Cham, 2020. doi: https://doi.org/10.1007/978-3-030-59638-5_7.

[8] Paulo Alves, Ronnie Paskin, Isabella Frajhof, Yang Miranda, João Jardim, Jose Cardoso, Eduardo Tress, Rogério Ferreira da Cunha, Rafael Nasser, and Gustavo Robichez. Exploring blockchain technology to improve multi-party relationship in business process management systems. In *Proceedings of the 22nd International Conference on Enterprise Information Systems*. SciTePress, 2020. doi: https://doi.org/10.5220/0009565108170825.

[9] Elli Androulaki, Artem Barger, et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. *CoRR*, abs/1801.10228, 2018. URL http://arxiv.org/abs/1801.10228.

[10] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. Modeling Bitcoin contracts by timed automata. In Axel Legay and Marius Bozga, editors, *Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 8711 of *Lecture Notes in Computer Science*, pages 7–22. Springer, Cham, 2014. doi: https://doi.org/10.1007/978-3-319-10512-3_2.

[11] Shaun Azzopardi, Gordon J. Pace, and Fernando Schapachnik. On observing contracts: Deontic contracts meet smart contracts. In *Legal Knowledge and Information Systems - JURIX 2018: The Thirty-first Annual Conference*, volume 313, pages 21–30. IOS Press, 2018. doi: https://doi.org/10.3233/978-1-61499-935-5-21.

[12] Shaun Azzopardi, Joshua Ellul, and Gordon Pace. Runtime monitoring processes across blockchains. In *9th IPM International Conference on Fundamentals of Software Engineering 2021 (FSEN 2021), Tehran, Iran*, 2021. URL http://www.cs.um.edu.mt/gordon.pace/Research/Papers/fsen2021.pdf.

[13] HMN Dilum Bandara, Xiwei Xu, and Ingo Weber. Patterns for blockchain data migration. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, EuroPLoP '20. Association for Computing Machinery, 2020. doi: https://doi.org/10.1145/3424771.3424796.

[14] Rafael Belchior, André Vasconcelos, Sérgio Guerreiro, and Miguel Correia. A survey on blockchain interoperability: Past, present, and future trends. *CoRR*, 2005.14282, 2020. URL https://arxiv.org/pdf/2005.14282.pdf.

[15] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, volume 8043 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2013. doi: https://doi.org/10.1007/978-3-642-40084-1_6.

[16] Sarah Bouraga. A taxonomy of blockchain consensus protocols: A survey and classification framework. *Expert Systems with Applications*, 168:114384, 2021. doi: https://doi.org/10.1016/j.eswa.2020.114384.

[17] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: An introduction, 2016. URL http://rgdoi.net/10.13140/RG.2.2.30487.37284.

[18] Bert-Jan Butijn, Damian A. Tamburri, and Willem-Jan van den Heuvel. Blockchains: A systematic multivocal literature review. *ACM Comput. Surv.*, 53(3), 6 2020. doi: https://doi.org/10.1145/3369052.

[19] Núria Casellas. *Legal Ontology Engineering: Methodologies, Modelling Trends, and the Ontology of Professional Judicial Knowledge*, volume 3 of *Law, Governance and Technology*. Springer Science & Business Media, 2011. doi: https://doi.org/10.1007/978-94-007-1497-7.

[20] Saoussen Cheikhrouhou, Slim Kallel, Nawal Guermouche, and Mohamed Jmaiel. The temporal perspective in business process modeling: A survey and research challenges. *Serv. Oriented Comput. Appl.*, 9(1):75–85, 2015. doi: https://doi.org/10.1007/s11761-014-0170-x.

[21] Olivia Choudhury, Nolan Rudolph, Issa Sylla, Noor Fairoza, and Amar Das. Auto-generation of smart contracts from domain-specific ontologies and semantic rules. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018. doi: https://doi.org/10.1109/Cybermatics_2018.2018.00183.

[22] 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. doi: https://doi.org/10.1007/s00287-019-01178-x.

[23] Christopher D. Clack, Vikram A. Bakshi, and Lee Braine. Smart contract templates: essential requirements and design options. *CoRR*, abs/1612.04496, 2016. URL https://arxiv.org/abs/1612.04496.

[24] Christopher D. Clack, Vikram A. Bakshi, and Lee Braine. Smart contract templates: foundations, design landscape and research directions. *CoRR*, abs/1608.00771, 2016. URL http://arxiv.org/abs/1608.00771.

[25] Morris R. Cohen. The basis of contract. *Harvard Law Review*, 46(4):553–592, 1933. ISSN 0017811X. URL http://www.jstor.org/stable/1331491.

[26] Flavio Corradini, Alessandro Marcelletti, Andrea Morichetta, Andrea Polini, Barbara Re, and Francesco Tiezzi. Engineering trustable choreography-based systems using blockchain. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 1470–1479, New York, NY, USA, 2020. Association for Computing Machinery. doi: https://doi.org/10.1145/3341105.3373988.

[27] Marcelo Corrales Compagnucci et al., editors. *Legal Tech, Smart Contracts and Blockchain*. Perspectives in Law, Business and Innovation. Springer, 2019. doi: https://doi.org/10.1007/978-981-13-6086-2.

[28] Sue E. S. Crawford and Elinor Ostrom. A grammar of institutions. *The American Political Science Review*, 89(3):582–600, 1995. doi: https://doi.org/10.2307/2082975.

[29] Deutsche Bahn AG. Kennzahlen 2019: Wussten Sie schon, dass..., 2019. URL https://www.deutschebahn.com/de/konzern/konzernprofil/zahlen_fakten/kennzahlen_2019-5058430.

[30] Deutsche Bahn AG. Tarifbekanntmachungen und Beförderungsbedingungen der Deutschen Bahn AG, 2021. URL https://www.bahn.de/p/view/home/agb/agb.shtml.

[31] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management*. Springer, 2nd edition, 2018. doi: https://doi.org/10.1007/978-3-662-56509-4.

[32] Jacob Eberhardt and Stefan Tai. On or off the blockchain? insights on off-chaining computation and data. In Flavio De Paoli et al., editors, *Service-Oriented and Cloud Computing*, volume 10465 of *Lecture Notes in Computer Science*, pages 3–15. Springer, Cham, 2017. doi: https://doi.org/10.1007/978-3-319-67262-5_1.

[33] Johann Eder, Euthimios Panagos, and Michael Rabinovich. Time constraints in workflow systems. In Matthias Jarke and Andreas Oberweis, editors, *Advanced Information Systems Engineering*, volume 1626 of *Lecture Notes in Computer Science*, pages 286–300. Springer, Berlin, Heidelberg, 1999. doi: https://doi.org/10.1007/3-540-48738-7_22.

[34] Ghareeb Falazi, Michael Hahn, Uwe Breitenbücher, and Frank Leymann. Modeling and execution of blockchain-aware business processes. *SICS Software-Intensive Cyber-Physical Systems*, 34(2-3):105–116, 2019. doi: https://doi.org/10.1007/s00450-019-00399-5.

[35] Mark Flood and Oliver Goodenough. Contract as automaton: The computational representation of financial agreements. *OFR Working Paper 15-04*, 2015. doi: https://doi.org/10.2139/ssrn.2538224.

[36] Julian Alberto Garcia-Garcia, Nicolás Sánchez-Gómez, David Lizcano, M. J. Escalona, and Tomás Wojdyński. Using blockchain to improve collaborative business process management: Systematic literature review. *IEEE Access*, 8:142312–142336, 2020. doi: https://doi.org/10.1109/ACCESS.2020.3013911.

[37] Bryan A. Garner. *Black's Law Dictionary*. Thomson/West, 2004. ISBN 978-0-314-15199-5.

[38] LM Goodman. Tezos—a self-amending crypto-ledger white paper, 2014. URL https://www.tezos.com/static/papers/white_paper.pdf.

[39] Guido Governatori, Florian Idelberger, Zoran Milosevic, Regis Riveret, Giovanni Sartor, and Xiwei Xu. On legal contracts, imperative and declarative smart contracts, and blockchain systems. *Artificial Intelligence and Law*, 26

(4):377–409, 2018. doi: https://doi.org/10.1007/s10506-018-9223-3.

[40] Cristine Griffo, João Paulo A. Almeida, and Giancarlo Guizzardi. Conceptual modeling of legal relations. In Juan C. Trujillo et al., editors, *Conceptual Modeling*, volume 11157 of *Lecture Notes in Computer Science*, pages 169–183. Springer, Cham, 2018. doi: https://doi.org/10.1007/978-3-030-00847-5_14.

[41] Ian Grigg. The Ricardian contract. In *First IEEE International Workshop on Electronic Contracting*, pages 25–31. IEEE, 2004. doi: https://doi.org/10.1109/WEC.2004.1319505.

[42] Stephan Haarmann. Estimating the duration of blockchain-based business processes using simulation. In Stefan Kolb and Christian Sturm, editors, *11th Central European Workshop on Services and their Composition (ZEUS)*, volume 2339, pages 24–31. CEUR-WS.org, 2019. URL http://ceur-ws.org/Vol-2339/paper5.pdf.

[43] Stephan Haarmann, Kimon Batoulis, Adriatik Nikaj, and Mathias Weske. DMN decision execution on the Ethereum blockchain. In *Advanced Information Systems Engineering (CAiSE)*, volume 10816 of *Lecture Notes in Computer Science*, pages 327–341. Springer, Cham, 2018. doi: https://doi.org/10.1007/978-3-319-91563-0_20.

[44] James Hazard and Helena Haapio. Wise contracts: Smart contracts that work for people and machines. In *Trends and Communities of Legal Informatics, 20th International Legal Informatics Symposium IRIS 2017*, pages 425–432, 2017. doi: https://doi.org/10.2139/ssrn.2925871.

[45] Mike Hearn. Corda: A distributed ledger, technical whitepaper, 2016. URL https://docs.corda.net/_static/corda-technical-whitepaper.pdf.

[46] Jonathan Heiss, Jacob Eberhardt, and Stefan Tai. From oracles to trustworthy data on-chaining systems. In *2019 IEEE International Conference on Blockchain*, pages 496–503, 2019. doi: https://doi.org/10.1109/Blockchain.2019.00075.

[47] Rinke Hoekstra, Joost Breuker, Marcello Di Bello, and Alexander Boer. The LKIF Core ontology of basic legal concepts. *2nd Workshop on Legal Ontologies and Artificial Intelligence Techniques*, pages 43–63, 2007. URL http://ceur-ws.org/Vol-321/paper3.pdf.

[48] Wesley Newcomb Hohfeld. Fundamental legal conceptions as applied in judicial reasoning. *The Yale Law Journal*, 26(8):710–770, 1917. doi: https://doi.org/10.2307/786270.

[49] Felix Härer. Decentralized business process modeling and instance tracking secured by a blockchain. In *Proceedings of the 26th European Conference on Information Systems (ECIS 2018)*, 2018. doi: https://doi.org/10.5281/zenodo.2585718.

[50] Tom Hvitved. *Contract Formalisation and Modular Implementation of Domain-Specific Languages*. PhD thesis, University of Copenhagen, 2012.

[51] Adnan Imeri, Nazim Agoulmine, and Djamel Khadraoui. Smart Contract modeling and verification techniques: A survey. In Francisco Moo-Mena and Elias Duarte, editors, *8th International Workshop on ADVANCEs in ICT Infrastructures and Services (ADVANCE 2020)*, pages 1–8, 2020. URL https://hal.archives-ouvertes.fr/hal-02495158.

[52] ISO/IEC 25010:2011. Systems and software engineering — Systems and software quality requirements and evaluation (SQuaRE) — System and software quality models. Standard, International Organization for Standardization (ISO), 2011.

[53] ISO/IEC 80000-2:2019. Quantities and units — Part 2: Mathematics. Standard, International Organization for Standardization (ISO), 2019.

[54] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets*. Springer, Berlin, Heidelberg, 2009. doi: https://doi.org/10.1007/b95112.

[55] Vandana Kabilan. Contract workflow model patterns using BPMN. In *Int. Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD), CAiSE*, volume 363 of *CEUR-WS.org*, 2005. URL http://ceur-ws.org/Vol-363/paper16.pdf.

[56] Vandana Kabilan and Paul Johannesson. Semantic representation of contract knowledge using multi-tier ontology. In *First Int. Conference on Semantic Web and Databases*, CEUR-WS.org, pages 378–397, 2003. URL https://dl.acm.org/doi/10.5555/2889905.2889930.

[57] Niclas Kannengießer, Michelle Pfister, Malte Greulich, Sebastian Lins, and Ali Sunyaev. Bridges between islands: Cross-chain technology for distributed ledger technology. In *Proceedings of the 53rd Hawaii International Conference on System Sciences*, 2020. doi: https://doi.org/10.24251/hicss.2020.652.

[58] Prabhakaran Kasinathan and Jorge Cuellar. Securing the integrity of workflows in IoT. In *Proceedings of the 2018 International Conference on Embedded Wireless Systems and Networks*, EWSN '18, page 252–257, USA, 2018. Junction Publishing. ISBN 9780994988621. URL https://dl.acm.org/doi/10.5555/3234847.3234908.

[59] Firas Al Khalil, Tom Butler, Leona O'Brien, and Marcello Ceci. Trust in smart contracts is a process, as well. In *Financial Cryptography and Data Security*, volume 10323 of *Lecture Notes in Computer Science*, pages 510–519, 2017. doi: https://doi.org/10.1007/978-3-319-70278-0_32.

[60] Philipp Klinger and Freimut Bodendorf. Blockchain-based cross-organizational execution framework for dynamic integration of process collaborations. In *15th International Conference on Wirtschaftsinformatik (WI)*, pages 893–908, 2020. doi: https://doi.org/10.30844/wi_2020_i2-klinger.

[61] Philipp Klinger, Long Nguyen, and Freimut Bodendorf. Upgradeability concept for collaborative blockchain-based business process execution framework. In Zhixiong Chen et al., editors, *Blockchain – ICBC 2020*, volume 12404 of *Lecture Notes in Computer Science*, pages 127–141. Springer, Cham, 2020. doi: https://doi.org/10.1007/978-3-030-59638-5_9.

[62] Christopher Klinkmüller, Alexander Ponomarev, An Binh Tran, Ingo Weber, and Wil van der Aalst. Mining blockchain processes: Extracting process mining data from blockchain applications. In Claudio Di Ciccio et al., editors, *Business Process Management: Blockchain and Central and Eastern Europe Forum*, volume 361 of *Lecture Notes in Business Information Processing*, pages 71–86. Springer, Cham, 2019. doi: https://doi.org/10.1007/978-3-030-30429-4_6.

[63] Julius Köpke, Marco Franceschetti, and Johann Eder. Balancing privity and enforceability of bpm-based smart contracts on blockchains. In Claudio Di Ciccio et al., editors, *Business Process Management: Blockchain and Central and Eastern Europe Forum*, volume 361 of *Lecture Notes in Business Information Processing*, pages 87–102. Springer, Cham, 2019. doi: https://doi.org/10.1007/978-3-030-30429-4_7.

[64] Kari Korpela, Jukka Hallikas, and Tomi Dahlberg. Digital supply chain transformation toward blockchain integration. In *Proceedings of the 50th Hawaii International Conference on System Sciences (2017)*, 2017. doi: https://doi.org/10.24251/hicss.2017.506.

[65] Jan Ladleif and Mathias Weske. A legal interpretation of choreography models. In Chiara Di Francescomarino et al., editors, *Business Process Management Workshops. BPM 2019*, volume 362 of *Lecture Notes in Business Information Processing*, pages 651–663. Springer, Cham, 2019. doi: https://doi.org/10.1007/978-3-030-37453-2_52.

[66] Jan Ladleif and Mathias Weske. A unifying model of legal smart contracts. In Alberto H. F. Laender et al., editors, *Conceptual Modeling*, volume 11788 of *Lecture Notes in Computer Science*, pages 323–337. Springer, Cham, 2019. doi: https://doi.org/10.1007/978-3-030-33223-5_27.

[67] Jan Ladleif and Mathias Weske. Time in blockchain-based process execution. In *24th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2020, Eindhoven, The Netherlands, October 5-8, 2020*, pages 217–226. IEEE, 2020. doi: https://doi.org/10.1109/EDOC49727.2020.00034.

[68] Jan Ladleif and Mathias Weske. Which event happened first? Deferred choice on blockchain using oracles. *CoRR*, abs/2104.10520, 2021. URL https://arxiv.org/abs/2104.10520.

[69] Jan Ladleif, Anton von Weltzien, and Mathias Weske. chor-js: A modeling framework for BPMN 2.0 choreography diagrams. In José Ignacio Panach et al., editors, *Proceedings of the ER Forum and Poster & Demos Session 2019, 38th International Conference on Conceptual Modeling (ER)*, volume 2469 of

CEUR WS Proceedings, pages 113–117, 2019. URL http://ceur-ws.org/Vol-2469/ERDemo02.pdf.

[70] Jan Ladleif, Mathias Weske, and Ingo Weber. Modeling and enforcing blockchain-based choreographies. In Thomas Hildebrandt et al., editors, *Business Process Management. BPM 2019*, volume 11675 of *Lecture Notes in Computer Science*, pages 69–85. Springer, Cham, 2019. doi: https://doi.org/10.1007/978-3-030-26619-6_7.

[71] Jan Ladleif, Christian Friedow, and Mathias Weske. An architecture for multi-chain business process choreographies. In Witold Abramowicz and Gary Klein, editors, *Business Information Systems*, volume 389 of *Lecture Notes in Business Information Processing*, pages 184–196. Springer, Cham, 2020. doi: https://doi.org/10.1007/978-3-030-53337-3_14.

[72] Jan Ladleif, Ingo Weber, and Mathias Weske. External data monitoring using oracles in blockchain-based process execution. In Aleksandre Asatiani et al., editors, *Business Process Management: Blockchain and Robotic Process Automation Forum, BPM 2020*, volume 393 of *Lecture Notes in Business Information Processing*, pages 67–81. Springer, Cham, 2020. doi: https://doi.org/10.1007/978-3-030-58779-6_5.

[73] Ronald M. Lee. A logic model for electronic contracting. *Decision Support Systems*, 4(1):27–44, March 1988. doi: https://doi.org/10.1016/0167-9236(88)90096-6.

[74] Sin Kuang Lo, Xiwei Xu, Mark Staples, and Lina Yao. Reliability analysis for blockchain oracles. *Computers & Electrical Engineering*, 83:106582, 2020. doi: https://doi.org/10.1016/j.compeleceng.2020.106582.

[75] Orlenys López-Pintado, Marlon Dumas, Luciano García-Bañuelos, and Ingo Weber. Interpreted execution of business process models on blockchain. In *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 206–215, 2019. doi: https://doi.org/10.1109/EDOC.2019.00033.

[76] Orlenys López-Pintado, Luciano García-Bañuelos, Marlon Dumas, Ingo Weber, and Alexander Ponomarev. Caterpillar: A business process execution engine on the Ethereum blockchain. *Software: Practice and Experience*, 49(7):1162–1193, 2019. doi: https://doi.org/10.1002/spe.2702.

[77] Qinghua Lu, An Binh Tran, Ingo Weber, Hugo O'Connor, Paul Rimba, Xiwei Xu, Mark Staples, Liming Zhu, and Ross Jeffery. Integrated model-driven engineering of blockchain applications for business processes and asset management. *CoRR*, 2020. URL http://arxiv.org/abs/2005.12685.

[78] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016. doi: https://doi.org/10.1145/2976749.2978309.

[79] 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*, 2018. URL https://static-curis.ku.dk/portal/files/194806456/fab18_submission_06.pdf.

[80] Kamran Mammadzada, Mubashar Iqbal, Fredrik Milani, Luciano García-Bañuelos, and Raimundas Matulevičius. Blockchain oracles: A framework for blockchain-based applications. In Aleksandre Asatiani et al., editors, *Business Process Management: Blockchain and Robotic Process Automation Forum*, volume 393 of *Lecture Notes in Business Information Processing*, pages 19–34. Springer, Cham, 2020. doi: https://doi.org/10.1007/978-3-030-58779-6_2.

[81] Anastasia Mavridou and Aron Laszka. Designing secure Ethereum smart contracts: A finite state machine based approach. *CoRR*, 2017. URL http://arxiv.org/abs/1711.09327.

[82] Anastasia Mavridou, Aron Laszka, Stachtiari Emmanouela, and Abhishek Dubey. Verisolid: Correct-by-design smart contracts for Ethereum. In *Proceedings of the 23nd International Conference on Financial Cryptography and Data Security (FC)*, volume 11598 of *Lecture Notes in Computer Science*. Springer, Cham, 2019. doi: https://doi.org/10.1007/978-3-030-32101-7_27.

[83] Jan Mendling, Ingo Weber, et al. Blockchains for business process management – challenges and opportunities. *ACM Transactions on Management Information Systems (TMIS)*, 9(1):4:1–4:16, 2018. doi: https://doi.org/10.1145/3183367.

[84] Roman Mühlberger, Stefan Bachhofner, Eduardo Castelló Ferrer, Claudio Di Ciccio, Ingo Weber, Maximilian Wöhrer, and Uwe Zdun. Foundational oracle patterns: Connecting blockchain to the off-chain world. In Aleksandre Asatiani et al., editors, *Business Process Management: Blockchain and Robotic Process Automation Forum*, volume 393 of *Lecture Notes in Business Information Processing*, pages 35–51. Springer, Cham, 2020. doi: https://doi.org/10.1007/978-3-030-58779-6_3.

[85] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL https://bitcoin.org/bitcoin.pdf.

[86] Jørgen Svennevik Notland, Jakob Svennevik Notland, and Donn Morrison. The minimum hybrid contract (MHC): Combining legal and blockchain smart contracts. In *Proceedings of the Evaluation and Assessment in Software Engineering*, EASE '20, page 390–397. Association for Computing Machinery, 2020. doi: https://doi.org/10.1145/3383219.3383275.

[87] OMG. Business Process Model and Notation (BPMN), Version 2.0.2, 2013. URL http://www.omg.org/spec/BPMN/2.0.2/.

[88] OMG. Meta Object Facility (MOF), Version 2.5.1, 2013. URL https://www.omg.org/spec/MOF/2.5.1/.

[89] Gordon J. Pace and Gerardo Schneider. Challenges in the specification of full contracts. In *Integrated Formal Methods*, volume 5423 of *Lecture Notes in Computer Science*, pages 292–306. Springer, Berlin, Heidelberg, 2009. doi: https://doi.org/10.1007/978-3-642-00255-7_20.

[90] Julien Polge, Jérémy Robert, and Yves Le Traon. Permissioned blockchain frameworks in the industry: A comparison. *ICT Express*, 2020. doi: https://doi.org/10.1016/j.icte.2020.09.002.

[91] Shaya Pourmirza, Sander Peters, Remco Dijkman, and Paul Grefen. A systematic literature review on the architecture of business process management systems. *Information Systems*, 66:43–58, 2017. ISSN 0306-4379. doi: https://doi.org/10.1016/j.is.2017.01.007.

[92] Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 4468 of *Lecture Notes in Computer Science*, pages 174–189. Springer, Berlin, Heidelberg, 2007. doi: https://doi.org/10.1007/978-3-540-72952-5_11.

[93] Wolfgang Reisig. *Petri Nets: An Introduction*. Springer, Berlin, Heidelberg, 1985. doi: https://doi.org/10.1007/978-3-642-69968-9.

[94] Nick Russell, Arthur HM Ter Hofstede, David Edmond, and Wil MP van der Aalst. Workflow data patterns. Technical Report FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.

[95] Nick Russell, Arthur HM Ter Hofstede, David Edmond, and Wil MP van der Aalst. Workflow resource patterns. Technical Report WP 127, Eindhoven University of Technology, Eindhoven, 2004.

[96] Nick Russell, Arthur HM Ter Hofstede, Wil MP Van Der Aalst, and Nataliya Mulyar. Workflow control-flow patterns: A revised view. *BPM Center Report BPM-06-22, BPMcenter. org*, pages 06–22, 2006.

[97] Naoto Sato, Takaaki Tateishi, and Shunichi Amano. Formal requirement enforcement on smart contracts based on linear dynamic logic. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 945–954, 2018. doi: https://doi.org/10.1109/Cybermatics_2018.2018.00181.

[98] Markus Schinle, Christina Erler, Philip Nicolai Andris, and Wilhelm Stork. Integration, execution and monitoring of business processes with chaincode. In *2nd Conference on Blockchain Research Applications for Innovative Networks and Services (BRAINS)*, pages 63–70, 2020. doi: https://doi.org/10.1109/BRAINS49436.2020.9223283.

[99] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006. doi: https://doi.org/10.1109/MC.2006.58.

[100] Nicolás Sánchez-Gómez, Jesus Torres-Valderrama, J. A. García-García, Javier J. Gutiérrez, and M. J. Escalona. Model-based software design and testing in blockchain smart contracts: A systematic literature review. *IEEE Access*, 8:164556–164569, 2020. doi: https://doi.org/10.1109/ACCESS.2020.3021502.

[101] Christian Sturm, Jonas Szalanczi, Stefan Schönig, and Stefan Jablonski. A lean architecture for blockchain based decentralized process execution. In Florian Daniel et al., editors, *Business Process Management Workshops. BPM 2018*, volume 342 of *Lecture Notes in Business Information Processing*. Springer, Cham, 2018. doi: https://doi.org/10.1007/978-3-030-11641-5_29.

[102] Christian Sturm, Jonas Szalanczi, Stefan Jablonski, and Stefan Schönig. Decentralized control: A novel form of interorganizational workflow interoperability. In *The Practice of Enterprise Modeling. PoEM 2020*, volume 400 of *Lecture Notes in Business Information Processing*, pages 261–276. Springer, Cham, 2020. doi: https://doi.org/10.1007/978-3-030-63479-7_18.

[103] Harry Surden. Computable contracts. *UC Davis Law Review*, 46(629), 2012. URL https://ssrn.com/abstract=2216866.

[104] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997. doi: https://doi.org/10.5210/fm.v2i9.548.

[105] Nick Szabo. A formal language for analyzing contracts, 2002. URL https://nakamotoinstitute.org/contract-language/.

[106] Takaaki Tateishi, Sachiko Yoshihama, Naoto Sato, and S. Saito. Automatic smart contract generation using controlled natural language and template. *IBM Journal of Research and Development*, 63(2/3):6:1–6:12, 2019. doi: https://doi.org/10.1147/JRD.2019.2900643.

[107] Wei-Tek Tsai, Ning Ge, Jiaying Jiang, Kevin Feng, and Juan He. Beagle: A new framework for smart contracts taking account of law. In *IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 134–13411, 2019. doi: https://doi.org/10.1109/SOSE.2019.00028.

[108] Chibuzor Udokwu, Aleksandr Kormiltsyn, Kondwani Thangalimodzi, and Alex Norta. The state of the art for blockchain-enabled smart-contract applications in the organization. In *2018 Ivannikov Ispras Open Conference (ISPRAS)*, pages 137–144, 2018. doi: https://doi.org/10.1109/ISPRAS.2018.00029.

[109] Ingo Weber, Xiwei Xu, Régis Riveret, Guido Governatori, Alexander Ponomarev, and Jan Mendling. Untrusted business process monitoring and execution using blockchain. In Marcello La Rosa et al., editors, *Business Process Management (BPM)*, volume 9850 of *Lecture Notes in Computer Science*, pages 329–347. Springer, Cham, 2016. doi: https://doi.org/10.1007/978-3-319-45348-4_19.

[110] Ingo Weber, Vincent Gramoli, Alex Ponomarev, Mark Staples, Ralph Holz, An Binh Tran, and Paul Rimba. On availability for blockchain-based systems. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 64–73, 2017. doi: https://doi.org/10.1109/SRDS.2017.15.

[111] Kevin Werbach and Nicolas Cornell. Contracts ex machina. *Duke Law Journal*, 67:313, 2017. URL https://ssrn.com/abstract=2936294.

[112] Mathias Weske. *Business Process Management*. Springer, Berlin, Heidelberg, 3rd edition, 2019. doi: https://doi.org/10.1007/978-3-662-59432-2.

[113] Maximilian Wöhrer and Uwe Zdun. Design patterns for smart contracts in the ethereum ecosystem. In *2018 IEEE International Conference on Blockchain*, pages 1513–1520, 2018. doi: https://doi.org/10.1109/Cybermatics_2018.2018.00255.

[114] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Technical report, Ethereum Project Yellow Paper, 2014.

[115] 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 *IEEE Intl. Conf. Software Architecture (ICSA)*, pages 243–252, 2017. doi: https://doi.org/10.1109/ICSA.2017.33.

[116] Xiwei Xu, Cesare Pautasso, Liming Zhu, Qinghua Lu, and Ingo Weber. A pattern collection for blockchain-based applications. In *23rd European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM, 2018. doi: https://doi.org/10.1145/3282308.3282312.

[117] Xiwei Xu, H. M. N. Dilum Bandara, Qinghua Lu, Dawen Zhang, and Liming Zhu. Understanding and handling blockchain uncertainties. In Zhixiong Chen et al., editors, *Blockchain – ICBC 2020*, Lecture Notes in Computer Science, pages 108–124. Springer, Cham, 2020. doi: https://doi.org/10.1007/978-3-030-59638-5_8.

[118] 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)*, pages 253–256, 2017. doi: https://doi.org/10.1109/ICSA.2017.22.

All links were last followed on April 28th, 2021. Further, all links in the body of the digital version of this thesis redirect to archived versions of the referenced websites via *Internet Archive* valid as of writing this thesis, if available.