Jan Ladleif | Mathias Weske

Which Event Happened First? Deferred Choice on Blockchain Using Oracles

Suggested citation referring to the original publication:
Frontiers in Blockchain 4 (2021), Art. 758169 pp. 1 - 16
DOI https://doi.org/10.3389/fbloc.2021.758169
ISSN 2624-7852

Journal article | Version of record

Secondary publication archived on the Publication Server of the University of Potsdam:
Zweitveröffentlichungen der Universität Potsdam : Reihe der Digital Engineering Fakultät 11

https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-550681
DOI: https://doi.org/10.25932/publishup-55068

frontiers in Blockchain

ORIGINAL RESEARCH
published: 26 October 2021
doi: 10.3389/fbloc.2021.758169

Which Event Happened First? Deferred Choice on Blockchain Using Oracles

Jan Ladleif and Mathias Weske*

Business Process Technology, Hasso Plattner Institute, University of Potsdam, Potsdam, Germany

First come, first served: Critical choices between alternative actions are often made based on events external to an organization, and reacting promptly to their occurrence can be a major advantage over the competition. In Business Process Management (BPM), such deferred choices can be expressed in process models, and they are an important aspect of process engines. Blockchain-based process execution approaches are no exception to this, but are severely limited by the inherent properties of the platform: The isolated environment prevents direct access to external entities and data, and the non-continual runtime based entirely on atomic transactions impedes the monitoring and detection of events. In this paper we provide an in-depth examination of the semantics of deferred choice, and transfer them to environments such as the blockchain. We introduce and compare several oracle architectures able to satisfy certain requirements, and show that they can be implemented using state-of-the-art blockchain technology.

Keywords: business processes, business process management, deferred choice, workflow patterns, blockchain, smart contracts, oracles, formal semantics

OPEN ACCESS

Edited by:
Claudio Di Ciccio,
Sapienza University of Rome, Italy

Reviewed by:
Giovanni Meroni,
Politecnico di Milano, Italy
Remo Pareschi,
University of Molise, Italy

****Correspondence:***
Jan Ladleif
jan.ladleif@hpi.de

Specialty section:
This article was submitted to Smart Contracts, a section of the journal Frontiers in Blockchain

Received: *13 August 2021*
Accepted: *05 October 2021*
Published: *26 October 2021*

Citation:
Ladleif J and Weske M (2021) Which Event Happened First? Deferred Choice on Blockchain Using Oracles. Front. Blockchain 4:758169. doi: 10.3389/fbloc.2021.758169

1 INTRODUCTION

A service failing due to a low-level system malfunction, an order from a customer being received, or a stock finally reaching the strike price—events are pervasive on every layer of abstraction in Business Process Management (BPM) (Weske, 2019), and reacting to them promptly significantly impacts the performance of an organization. An important pattern in this context is deferred choice, which describes situations in which an exclusive branching decision in a process instance depends on which one of a set of events occurs first (Russell et al., 2006). Deferred choice resembles a race between events, and only one can win.

Consider, for instance, a business process describing the purchase and usage of a train ticket, shown in **Figure 1** as a Business Process Model and Notation (BPMN) collaboration diagram. The customer books a ticket, optionally using their subscription-based discount card, and takes the train as expressed by the message event e_t triggered by the Railway Infrastructure Manager (RIM). There are several alternative paths after the event-based gateway g—the canonical representation of deferred choice in BPMN (OMG, 2013). The customer may cancel their ticket (e_c), severe weather may prohibit the train's departure (e_w), or the customer's discount card may expire (e_d). In all cases, the ticket is eventually voided.

It is the job of the process engine to be the judge of this race. To this end, a process engine monitors event sources continuously using a wide range of techniques like querying and polling public interfaces, repeatedly evaluating event conditions, or subscribing to news channels. Two goals are paramount: To 1) correctly pick the winner of the deferred choice, avoiding the

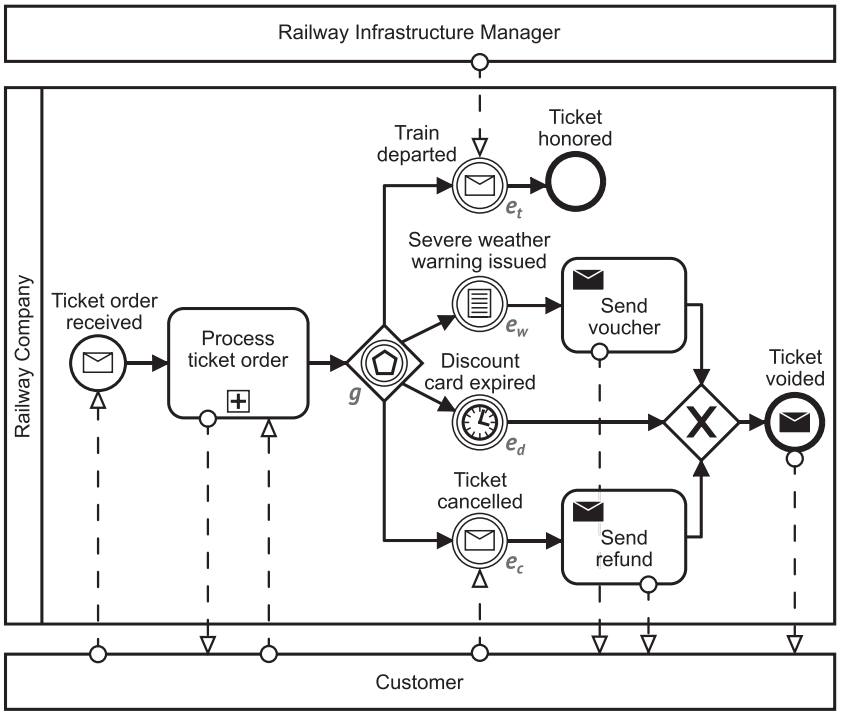

FIGURE 1 | Running example of a deferred choice scenario.

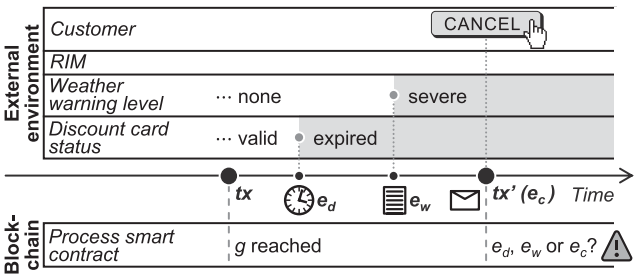

FIGURE 2 | Indeterminate behavior of a process smart contract due to temporal gaps between transactions.

spurious execution of activities, and 2) to do so in a timely manner, avoiding delays which have a negative impact on the business.

Recent approaches at implementing core aspects of process engines within smart contracts—programs whose code and state is stored on blockchains and which are executed within discrete transactions (Xu et al., 2017)—face severe issues trying to achieve those goals, however. While the integrity and immutability guarantees of blockchains may provide clear benefits (Mendling et al., 2018), they cause a peculiar execution environment: Smart contracts are neither constantly running, instead laying dormant outside of transactions (non-continuity property), nor can they directly interact with external services for integrity reasons (isolation property) (Xu et al., 2018).

The impact of these properties when trying to implement deferred choice within smart contracts is considerable, as is illustrated in **Figure 2**. Within a transaction *tx* the process execution reaches the gateway *g* (see **Figure 1**) and the deferred choice is started. Since no event can immediately be detected, the transaction finishes. While the smart contract lies dormant, the customer's discount card expires, and the meteorological service issues a severe weather warning. When the customer decides to cancel the ticket by explicitly sending a transaction tx' the smart contract is woken up again, and is able to detect all three events: The transaction tx' itself leads to the occurrence and detection of e_c. The temporal event e_d is detected since the associated deadline has passed. Lastly, an oracle—patterns used to somewhat circumvent the isolation property of smart contracts (Xu et al., 2018)—can be employed to acquire a snapshot of the current weather warning level and detect e_w.

The example reveals two major challenges: For one, it is not clear which event occurred first and is the correct winner of the race. State-of-the-art oracles only provide current data (Xu et al., 2018), and the smart contract is not able to decide whether e_w occurred before e_d or e_c. Second, the deferred choice is not resolved in a timely manner, since there could be an arbitrary delay between *tx* and tx'. Thus, both central tasks of the process engine concerning deferred choice are not fulfilled.

As a consequence, support for deferred choice patterns is essentially absent in current blockchain-based process execution approaches. In this paper, we thus consider one central research question: How can a correct and timely execution of general

deferred choice patterns be achieved from within isolated and non-continuous smart contracts? We particularly consider practical deferred choice scenarios, in which a heterogenous set of event types, both explicit and implicit, needs to be accounted for. Any proposed solution will be judged using three metrics:

- Correctness, i.e., whether the deferred choice semantics are correctly transferred.
- Immediacy, i.e., whether there are delays in resolving the deferred choice after an event has occurred.
- Cost, i.e., whether the approach can be implemented in a feasible and economical fashion.

This paper is built on some of our previous work, in which we have shown that there is minimal or non-existing support for events depending on temporal constraints (Ladleif and Weske, 2020) or external data sources (Ladleif et al., 2020) in state-of-the-art blockchain-based process engines. In particular, we adopt the latter work's idea of novel oracle patterns allowing access to historical values of external data sources and those supporting subscriptions to new data, which is underpinned by a novel formal semantics of deferred choice. We refine these ideas by providing concrete architectures and interface specifications in this paper, and combine everything to provide support for heterogenous deferred choice patterns. Ideally, our solution can provide a blueprint for future developments in blockchain-based process engines, enhancing their level of support and acceptance.

The paper is structured as follows: We first introduce preliminaries from the fields of BPM and blockchain technology in **Section 2**, and give an overview of related work in **Section 3**. We start our approach with the definition of a formal execution semantics for deferred choice in both continual and non-continual execution environments in **Section 4**. We then propose generic oracle architectures to implement the semantics in **Section 5**, and provide a concrete prototype in **Section 6** which forms the basis of an evaluation in **Section 7**. We discuss our results and conclude in **Section 8**.

2 PRELIMINARIES

In this section, preliminary knowledge about BPM and blockchain technology will be introduced.

2.1 Business Process Management and Events

Businesses are driven by recurring processes, in which activities are performed to reach a certain business goal. BPM is concerned with making these processes palpable and support them during their entire lifecycle: from initial identification to structured modeling and execution using a process engine and subsequent optimization (Weske, 2019).

Events are an essential aspect of business processes and refer to "points in time" (Weske, 2019, p. 85) at which something happens, in contrast to activities with a particular duration. Events can take many shapes. The *de facto* industry standard for business process models, BPMN, for example, includes more than 10 types of events (OMG, 2013). From error events, which represent system errors interrupting the regular process execution, to compensation events, which start attempts at rolling back certain activities—there are many options to represent critical business scenarios.

During the execution of a process, a process engine has to detect events as they occur to correctly apply their effects in a timely fashion. This is especially complex for events occurring external to the process engine. Common examples are events caused by explicit external actions, like receiving an order or inquiries from customers within messages. External events can also be caused more implicitly based on properties of the larger execution environment of the process, such as detecting a condition becoming satisfied, e.g., a stock reaching a specific price (Russell et al., 2006). Timer events also fall into this category and rely on the current time of the environment (Eder et al., 1999), allowing absolute or relative temporal constraints to be specified (Cheikhrouhou et al., 2015).

External events are the core building blocks of one of the fundamental patterns in workflow modeling and BPM: deferred choice (Russell et al., 2006). Deferred choice describes situations in which the exclusive choice between alternative execution branches of a process depends on the operating environment, i.e., which of a set of external events is detected first. As these events are caused by actions or circumstances outside the influence of the process engine, the choice is deferred until the first event is detected—essentially modeling a "race condition where the first [e]vent that is triggered wins" (OMG, 2013, p. 298).

In their seminal research on workflow patterns, Russell et al. describe the semantics of deferred choice using colored Petri nets (Russell et al., 2006). The "moment of choice", that is waiting for an event to occur, is modeled as a place, and each alternative path as an outgoing transition. Thus, only one branch may be taken per token. Most subsequent literature to this day adopts these semantics: Dijkman et al. (2008), for instance, use the Petri net abstraction for analysis and verification purposes. The BPMN standard itself describes a similar token system, but each outgoing branch initially receives a token and all but one are withdrawn after the choice is made (OMG, 2013).

2.2 Blockchain and Business Process Management

Advancing from its roots in cryptocurrencies (Nakamoto, 2008), blockchain has since emerged as a dedicated software platform which especially lends itself to business processes (Weber et al., 2016).

2.2.1 Blockchain Technology

Blockchains are distributed ledgers which store and process data and transactions. Smart contracts are programs whose code and state is persisted as data on the blockchain, and which can be called using transactions targeting their exposed functions (Xu

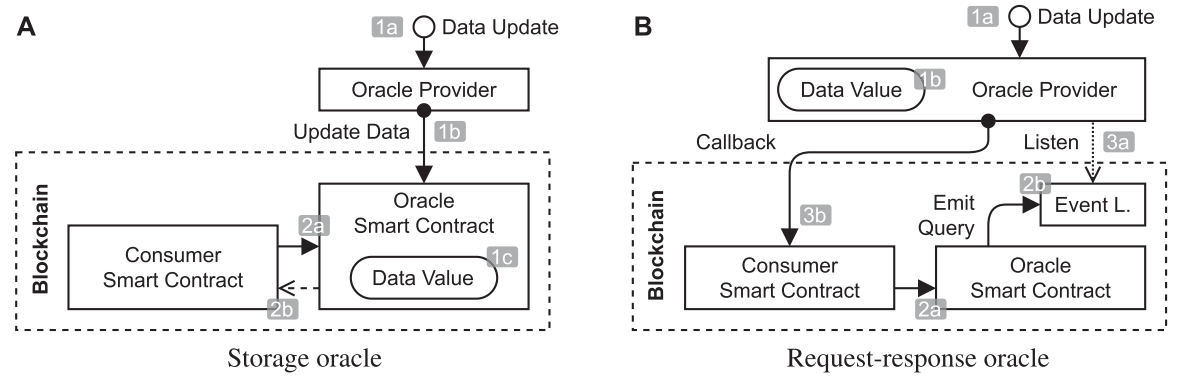

FIGURE 3 | Architecture and behavior of the storage and request/response oracles.

et al., 2017). The result of these transactions comes into effect during mining, a process in which transactions are ordered, executed, and bundled into a new block cryptographically linked to its predecessor.

Smart contracts and interactions with them enjoy all the benefits of blockchain technology: Their integrity can be easily validated, the origin of transactions can not be repudiated, and the smart contract state can not be forged (Xu et al., 2017). This makes them a valuable system component in low trust but high stakes scenarios involving several organizations, such as business process collaborations.

As a consequence, however, smart contracts need to adhere to several restrictions caused by the blockchain's idiosyncratic properties. For one, they operate in an isolated closed-world environment for integrity and traceability reasons, meaning they can not access any data or service outside the blockchain itself. Secondly, smart contracts are inherently passive and are only executed within discrete transactions during the mining process. This lends them a kind of transaction-driven or non-continual character, as they will always be "paused" between transactions and can not implement ongoing behavior such as busy-waiting.

2.2.2 Oracles

In practice, oracles are used to somewhat escape the isolation of blockchains and connect smart contracts to external entities (Xu et al., 2018). They generally consist of two components: an off-chain oracle provider, which is not subject to the blockchain's restrictions and can freely access data, services, and entities; and a publicly known oracle smart contract, which serves as a link between the consumers and the oracle provider. A recent survey identified two major and one minor type of oracles in research and practice (Al-Breiki et al., 2020):

The storage oracle (see **Figure 3A**) provides synchronous data access by storing current data on the blockchain. On each data update (1a) a transaction is sent (1b) to update a storage variable (1c) in the oracle smart contract. Consumers can then query the oracle smart contract (2a), which directly responds with the last known data (2b).

The request/response oracle (see **Figure 3B**) provides asynchronous access to external data, avoiding the use of on-chain storage. Instead, the oracle provider keeps track of the current data, for example by getting updates (1a) and storing them locally (1b). Consumer smart contracts can call the oracle smart contract (2a), which emits the query in an event using the blockchain's event layer (2b). The oracle provider picks up this event (3a), and sends the data to the consumer smart contract in a new transaction (3b).

Note that the event layer in this context refers to a capability of Ethereum and related blockchains to store additional pieces of data, so-called events, within blocks (Wood, 2014). Nodes observing the blockchain can then quickly scan new blocks for the information they are interested in. Thus, despite the name, they are not equivalent to events as defined above in the context of BPM. Rather, they can be thought of as additional output of transactions.

Lastly, there are publish/subscribe oracles. The goal is to access data "that is expected to change" (Al-Breiki et al., 2020), which is achieved through on-chain or off-chain flags which are manually polled. Note that while we use the same name for our own oracle architecture introduced in previous work (Ladleif et al., 2020) and properly specified in this paper (see **Section 5**), the patterns are fundamentally different. Publish/subscribe oracles as per Al-Breiki et al. are insufficient for implementing non-continual semantics using the publish/subscribe strategy (see Definition 10), since the active notification of changes is a prerequisite—no manual polling must be involved.

2.2.3 Blockchain-Based Process Engines

The potential synergy of blockchain technology and BPM was acknowledged early on (Mendling et al., 2018), shortly after first work was conducted to use smart contracts for secure and traceable process execution (Weber et al., 2016). This has spawned an impressive amount of research, especially regarding collaborative processes which benefit considerably from the blockchain's security properties (Garcia-Garcia et al., 2020).

In most approaches, one or more smart contracts keep track of the state of a process instance. Actions within the process are funneled through the smart contract using transactions, which can directly advance the state of the process instance accordingly

or be rejected if they do not conform to the process specification. In any case, the blockchain provides a tamper-proof audit log (Weber et al., 2016). Such process smart contracts need to implement the correct semantics of the original process model specification. Whether this is achieved using code generation (Weber et al., 2016; López-Pintado et al., 2019a; Ladleif et al., 2019) or interpretation (López-Pintado et al., 2019b) is secondary, as long as each source process concept is mapped to an equivalent smart contract concept.

In practice, this mapping is a particular problem for external events and the deferred choice pattern—which we will elaborate on in the remainder of this paper.

3 RELATED WORK

In this paper, we consider the semantics of deferred choice in formal terms, and propose concrete oracle architectures and provide a blockchain-based implementation. Accordingly, we refer to related work from these areas.

3.1 Deferred Choice Semantics

Part of our contribution in this paper is the specification of a formal definition and semantics of deferred choice in processes, taking into account the external execution environment and various heterogenous event types (see **Section 4**). Existing semantics specifications of deferred choice based on Petri nets (Russell et al., 2006; Dijkman et al., 2008) mostly gloss over the latter points, abstracting away from the mechanics of detecting events. Other literature, often focused on collaborations, exhibits similar limitations, and only consider explicit message events and their detection (Kheldoun et al., 2017; Corradini et al., 2018; Houhou et al., 2019). There is no notion of transactions or external data sources, and no discussion of the interplay with implicit—timer, conditional, or otherwise—events.

Kossak et al. (2012) provide the most detailed discussion of the semantics of deferred choice as modeled using event-based gateways in BPMN. Their main focus, however, is on instantiating event-based gateways and their inconsistencies, which are a special flavor of deferred choice that is not immediately applicable to this paper.

Overall, we find that existing literature on the semantics of deferred choice is mostly focused on verification, and if external events are considered at all this is mostly limited to messages. We thus provide a novel perspective that is especially useful for non-continual environments.

3.2 Deferred Choice on Blockchain

Numerous approaches at blockchain-based process execution have emerged, ranging from limited prototypes to powerful process engines with support for a wide range of business process constructs (see **Section 2.2**). However, deferred choice in its entirety is rarely among those supported constructs.

López-Pintado et al. (2019a) implement deferred choice only for internal as well as message events mapped to transactions in their Caterpillar engine. This circumvents the problem of gaps in perception since no implicit external events ever occur outside of transactions. Corradini et al. (2020) follow a choreography-based approach and all events competing in a deferred choice are directly mapped to transactions, which leads to the choice being resolved by the ordering of the transactions themselves. Adams et al. (2020), on the other hand, propose the notion of a "blockchain-integrated Business Process Management System (BPMS)" in which the blockchain is only used for storing "key contractual terms". That is, the local BPMS of the participants are responsible for executing workflow patterns including deferred choice, although it is not clear how tamper-proof enforcement using the smart contract can then be achieved.

Approaches like Lorikeet (Lu et al., 2021) and others (Weber et al., 2016; García-Bañuelos et al., 2017; Madsen et al., 2018; López-Pintado et al., 2019b; Klinger and Bodendorf, 2020; Azzopardi et al., 2021) likewise suggest some support for deferred choice for messages and internal events, but do not discuss related issues in detail. This uncertainty is exacerbated by the fact that prototypical implementations are rarely publicly available.

Still, even though deferred choice does not seem to have been a focus in any existing work, some do consider more types of events. In our own work, we implement conditional events to monitor local process data (Ladleif et al., 2019). Some approaches support constructs which can be used to emulate the behavior of external events, like service tasks in Caterpillar (López-Pintado et al., 2019a) or on-chain asset registries in Lorikeet (Lu et al., 2021). Weber et al. (2016) discuss connecting to external services using a dedicated trigger component, which could also assume the role of a request/response oracle. Similarly, many approaches including all of the above allow the inclusion of custom script annotations within their source models, which could potentially be used to access oracles non-natively and emulate resolution of deferred choice on a process level. In this paper, though, we strive for a native support for events.

Lastly, temporal constraints and timer events are largely absent in blockchain-based process engines due to the platform's inherent difficulties in these regards (Ladleif and Weske, 2020). They are sometimes mentioned (Weber et al., 2016; Ladleif et al., 2019; Klinger and Bodendorf, 2020), but never discussed in detail. Abid et al. (2020) provide a notable exception and extend Caterpillar with several notions of timer events, albeit not in a deferred choice setting. To this end, they use the block timestamps found on Ethereum blockchains to gauge the transaction timestamp, and implement time checks directly in the contract logic. In our prototype, we have used a similar approach and connected it with the timed event detection of conditional events (see **Section 4**).

In summary, deferred choice has not been the focus of any approach at blockchain-based process execution yet. Thus, we find that our work significantly extends on existing research in numerous fields, and is the first to provide an end-to-end vision of the usage of the deferred choice pattern on blockchain.

4 DEFERRED CHOICE EXECUTION SEMANTICS

To gauge the impact of non-continual execution platforms like blockchains, we first introduce a formal semantics of deferred choice from a continual perspective. It comprises of two layers as suggested in literature (Russell et al., 2006): the operating environment in which events explicitly occur or whose properties make events implicitly occur, and the system implementing the deferred choice pattern and detecting those events. We then adapt the semantics to the notion of transactions to illustrate the issues introduced by the isolation and non-continuity properties of blockchain.

4.1 Operating Environment

A deferred choice represents a local decision based on external factors outside the influence of the process engine (see **Section 2.1**), which are summarized as the environment:

Definition 1. (environment). The (operating) environment encapsulates all actors, systems, and information external to the process engine which are necessary to execute a deferred choice. In particular, this includes a set D of external variables.

Clearly, the environment is tailored to an individual application of the deferred choice pattern. For instance, the environment of the deferred choice g in the train ticket process (see **Figure 1**) comprises of two external actors: the customer and the RIM. There also has to be an external variable, say $D = \{d_w\}$, holding the current weather warning level, which would be stored within a database at the meteorological service. Lastly, the current time is logically part of the environment, and is used to keep track of the discount card expiration.

To keep our definitions concise, we assume in the following that we consider exactly one arbitrary but fixed deferred choice and its associated environment. Likewise, all examples will refer to the deferred choice g from our motivating example (see **Section 1**).

While explicit events are usually actively triggered by external actors, implicit events rely on certain environment properties at a specific point in time. For example, conditional events are bound to the values of an external variable, and timer events to the current environment time, all subsumed in the environment state:

Definition 2. (environment state). Let $\mathbb{D}$ be the domain of all possible values of external variables. Then an environment state $s = (t, \nu)$ is a tuple consisting of a physical timestamp $t \in \mathbb{N}$ and a valuation function ν: $D \to \mathbb{D}$ assigning a value to each variable. The set of all environment states is called $\mathcal{ES}$.

We use the natural numbers $\mathbb{N}$ as timestamps, which is a common abstraction found in information systems, and results in a discrete and uniform time domain. Note that the timestamp represents *physical* time rather than logical time, i.e., it is tied to some agreed-upon physical notion of time. Naturally, the state of the environment changes and evolves as time passes:

Definition 3. (successors). *Let* $s = (t, \nu), s' = (t', \nu') \in \mathcal{ES}$ *be environment states. Then* s' *is a successor of* s, *or* $s \to s'$, *iff* $t' = t + 1$.

An example trace of such environment states is shown in the upper part of **Table 1**, e.g., $s_1 = (t_1, \nu_1)$ with $t_1 = 73$ and $\nu_1(d_w) = 0$. The timestamps grow by one in each successive state, and the valuation function changes unpredictably.

4.2 Deferred Choice

The structure of a deferred choice is simple:

Definition 4. (deferred choice). A deferred choice is made between a non-empty set E of external events.

These events can be both explicit and implicit. In the train ticket example, for instance, the deferred choice g is given by $E = \{e_b, e_w, e_d, e_c\}$.

It is helpful to think of a deferred choice as a "race" with a single winner. Like for all races, there are two important milestones: when it starts and when it ends. A deferred choice starts or is *activated* when it is encountered during process execution, e.g., gateway g is reached; and ends once a winning event is detected, e.g., the expiration of the discount card e_d. The detection of events is contingent on the current environment state as well as the one at activation:

Definition 5. (event detection)**.** Given a deferred choice E, $\det: E \times \mathcal{ES} \times \mathcal{ES} \to \{true, false\}$ is the Boolean detection function with $\det(e, s_a, s) = true$, iff an event e can be detected in environment state s assuming activation happened in s_a.

In extension, $E_{\det}: \mathcal{ES} \times \mathcal{ES} \to 2^E$ with $E_{\det}(s_a, s) := \{e \in E \mid \det(e, s_a, s)\}$ is the set of all implicit events which can be detected under these circumstances.

The concrete definition of *det* depends on the type of event. Several examples can be seen in the middle part of **Table 1**. For the implicit events, expressions over the environment states can be given: $\det(e_d, s_a, s) \hat{=} t \geq 76$ means that the discount card expires at timestamp 76; and $\det(e_w, s_a, s) \hat{=} \nu(d_w) \geq 2$ means that the severe weather warning event e_w is detected as the corresponding external variable d_w reaches the value 2. Thus, both these events can be detected in s_5 assuming activation at s_1: $E_{det}(s_1, s_5) = \{e_d, e_w\}$. Explicit events, on the other hand, are based on concrete actions in the environment, e.g., messages sent by external actors, and follow no discernible pattern.

4.3 Continual Execution Semantics

In a race, a winner is determined immediately upon their reaching of the goal. Achieving this behavior for deferred choice (see **Section 2.1**) requires a continual observation of the environment to detect events as soon as possible. To express this formally, we propose a state transition system based on the evolution of the deferred choice alongside the environment state:

Definition 6. (deferred choice state). Given a deferred choice E, a deferred choice state $(s_a, s, e) \in \mathcal{ES} \times \mathcal{ES} \times (E \cup \{nil\})$ is a tuple

TABLE 1 | Operating environment, event detections and example deferred choice traces for the train ticket scenario.

Environment trace															
Environment states s_i	$\cdots$	$\rightarrow$	s_1	$\rightarrow$	s_2	$\rightarrow$	s_3	$\rightarrow$	s_4	$\rightarrow$	s_5	$\rightarrow$	s_6	$\rightarrow$	$\cdots$
Timestamp t_i	$\cdots$		73		74		75		76		77		78		$\cdots$
Valuation $\nu_i(d_w)$	$\cdots$		0		1		1		1		2		2		$\cdots$
Event detections															
$det(e_d, s_1, s_i) \doteq 76 \leq t_i$	$\cdots$								true		true		true		$\cdots$
$det(e_w, s_1, s_i) \doteq \nu_i(d_w) \geq 2$	$\cdots$										true		true		$\cdots$
$det(e_c, s_1, s_i)$	$\cdots$												true		$\cdots$
$det(e_t, s_1, s_i)$	$\cdots$														$\cdots$
Deferred choice traces															
(a) Continual	$\mathcal{CS}_0 \ni (s_1, s_1, nil) \rightsquigarrow (s_1, s_2, nil) \rightsquigarrow (s_1, s_3, nil) \overset{\top}{\rightsquigarrow} (s_1, s_4, e_d) \in \mathcal{F}$														
(b) Non-continual, history	$\mathcal{CS}_0 \ni (s_1, s_1, nil) \overset{\top}{\rightsquigarrow}$												$(s_1, s_6, e_d) \in \mathcal{F}$		
(c) Non-continual, pub/sub	$\mathcal{CS}_0 \ni (s_1, s_1, nil) \overset{\top}{\rightsquigarrow} (s_1, s_2, nil) \overset{\top}{\rightsquigarrow}$										$(s_1, s_5, e_d) \in \mathcal{F}$				

with s_a the environment state at activation, s the last observed environment state, and e the winning event or *nil*, if no event has won yet. The set of all such states is called $\mathcal{CS}$.

The initial states of the transition system are determined by the environment state when the deferred choice is activated and the race starts:

Definition 7. (initial states). The set $\mathcal{CS}_0 \subseteq \mathcal{CS}$ of all initial states of a deferred choice E is given by

$$\mathcal{CS}_0 := \{(s, s, e) \mid s \in \mathcal{ES} \wedge \quad e \in E_{det}(s, s) \vee (E_{det}(s, s) = \varnothing \wedge e = nil))\}$$

Notably, an initial state might already be the end of the race: If events can be detected immediately, one of them must be chosen. There is no sense of priority; for example (see **Table 1**), (s_5, s_5, e_d) and (s_5, s_5, e_w) are the valid initial states when starting in s_5, but (s_1, s_1, nil) is the only one when starting in s_1. The race ends when a winner is found:

Definition 8. (final states). $\mathcal{F} := \{(s_a, s, e) \in \mathcal{CS} \mid e \neq nil\}$ is the set of all final deferred choice states, i.e., those in which an event has won.

Using these notions, we can finally define the operational semantics of deferred choice using an unlabeled terminal transition system with initial states (Plotkin, 1981):

Definition 9. (continual transition system). Given a deferred choice E, the transition system $\langle \mathcal{CS}, \rightsquigarrow, \mathcal{CS}_0, \mathcal{F} \rangle$ with a transition relation $\rightsquigarrow \subseteq \mathcal{CS} \times \mathcal{CS}$ such that

$$\begin{aligned} &(s_a, s, nil) \rightsquigarrow (s_a, s', e) \\ \Leftrightarrow \quad & s \rightarrow s' \wedge && \text{(i)} \\ & (e \in E \Rightarrow e \in E_{\text{det}}(s_a, s')) \wedge && \text{(ii)} \\ & (e = nil \Rightarrow E_{\text{det}}(s_a, s') = \varnothing) && \text{(iii)} \end{aligned}$$

describes its continual execution semantics.

The transition relation $\rightsquigarrow$ includes several constraints. For one, (i) the two referenced environment states need to be direct successors. There can be no gap in time between them, expressing the continual nature of transitions. Further, (ii) if an event wins, it must be among those detected in the new environment state s'. Lastly, (iii) the winner may only remain undecided if no event could have been detected.

The bottom part of **Table 1** shows an example trace (a) of the continual semantics, starting with the state $(s_1, s_1, nil) \in \mathcal{CS}_0$. The state evolves alongside the environment, until e_d is detected and the transition system arrives in the final state $(s_1, s_4, e_d) \in \mathcal{F}$.

4.4 Non-Continual Execution Semantics

The continual execution semantics works under the assumption that each environment state is observed: Deadlines being reached or conditions becoming satisfied will immediately be registered, and corresponding events be detected.

However, this is not the case in a non-continual environment like the blockchain. Each change of the deferred choice state, that is activating and transitioning to new states with $\rightsquigarrow$, needs to be contained within a blockchain transaction. Time passes between transactions, in which the smart contract storing the state lies dormant. This directly leads to environment states being missed, violating rule (i) of the continual transition relation $\rightsquigarrow$.

Thus, $\rightsquigarrow$ needs to be adapted. Rule (i) needs to be relaxed to allow for gaps, and the other rules modified to cope with those gaps. In a first step, we reconsider the event detection function *det*. Instead of just checking *whether* an implicit event may be detected in a single environment state, we extend it to return the time at which an event was first detected—that is, when it crossed the finish line in the race:

Definition 10. (timed event detection). Let $\top \in \mathbb{N}$ be a fixed and sufficiently large timestamp arbitrarily far in the future that will realistically never be reached in the environment.

Given a deferred choice E, the function det_T: $E \times \mathcal{ES} \times \mathcal{ES} \rightarrow \mathbb{N}$ determines for $\text{det}_T(e, s_a, s)$ the earliest timestamp at

which e could have been detected starting from the environment state s_a up to s. If no such timestamp exists, $\top$ is returned as an indicator that the event could not have been detected.

In extension, $t_T: \mathcal{ES} \times \mathcal{ES} \to \mathbb{N}$ *with* $t_T(s_a, s) := \min\{\det_T(e, s_a, s) \mid e \in E\}$ is the earliest detection time of any event, or $\top$ if none was detected at all.

For example, in the train ticket scenario (see **Table 1**), the discount card first expired at timestamp 76, leading to, e.g., $\det_T(e_d, s_1, s_6) = 76$. It is also the first event to occur, thus $t_T(s_1, s_6) = 76$ holds as well. Assuming that such a function exists—which we will qualify in the next section—, we can devise of a new transition system:

Definition 11. (non-continual transition system). Given a deferred choice E, the transition system $\langle \mathcal{CS}, \overset{T}{\nrightarrow}, \mathcal{CS}_0, \mathcal{F} \rangle$ with a transition relation $\overset{T}{\nrightarrow} \subseteq \mathcal{CS} \times \mathcal{CS}$ such that

$$
\begin{aligned}
&(s_a, s, nil) \overset{T}{\nrightarrow} (s_a, s', e) \\
&\Leftrightarrow \quad t < t' \wedge && \text{(i)} \\
&\qquad (e \in E \Rightarrow det_T(e, s_a, s') = t_T(s_a, s') \neq \top) && \text{(ii)} \\
&\qquad (e = nil \Rightarrow t_T(s_a, s') = \top) && \text{(iii)}
\end{aligned}
$$

describes its non-continual execution semantics.

The non-continual transition relation $\overset{T}{\nrightarrow}$ only requires that s' is after s and not a direct successor anymore in rule (i)—which allows for the mentioned gaps. Rule (ii) expresses that if an event is chosen as the winner, it must have been detected and no event may have been detected earlier. Rule (iii) checks that the winner remains undecided only if no event could have been detected at all.

Speaking in terms of the race analogy, the non-continual semantics basically corresponds to the referee determining the time each participant has finished after the race, and retroactively deciding on the winner. **Table 1** shows example traces (b) and (c) for this behavior. In trace (b), we directly transition from the initial state (s_1, s_1, nil) to (s_1, s_6, e_d), correctly identifying e_d as the winner. Trace (c) arrives at the same conclusion.

4.5 Timed Event Detection on Blockchain

While the non-continual transition relation $\overset{T}{\nrightarrow}$ ostensibly solves the issues of deferred choice on blockchain, it relies on a major assumption: that a timed detection function det_T exists. The actual definition of such a function is needed for an implementation, however, and may be non-obvious depending on the type of event that is concerned. In the scope of this paper, we consider three types of events:

4.5.1 Message Events

In blockchain-based process execution, message events usually get translated to transactions as shown in **Figure 2**. The detection of a message event is then equivalent to the corresponding transaction being mined, which can easily be retrieved by the smart contract. det_T for message events thus comes down to determining whether the current transaction directly corresponds to a message.

4.5.2 Timer Events

Since the exact deadline for absolute and delay for relative timers is known, it is easy to retroactively determine when they should have been detected first. Given, for example, a relative timer event e with a delay δ and a detection function of the form $\det(e, s_a, s) \hat{=} t_a + \delta \leq t$, we can directly derive $\det_T$:

$$
\det_T(e, s_a, s) \hat{=} \begin{cases} t_a + \delta & \text{if } t_a + \delta \leq t \\ \top & \text{otherwise} \end{cases}
$$

If the delay has passed since activation, it has done so exactly $t_a + \delta$. For absolute timer events, $\det_T$ looks similar.

4.5.3 Conditional Events

The valuation function ν may change arbitrarily between environment states. There is no way to deduce the intermediate values of external variables from the environment states at activation and transitioning alone. Thus, det_T can not be specified for conditional events without additional assumptions. We suggest two approaches to this end:

The history approach assumes the intermediate environment states $s_a = s_1 \to s_2 \to \cdots \to s_n = s$ and thus all valuations of the external variables are available. Using a simple search, one can then find the earliest timestamp at which the continual detection function det returned *true*:

$$
\det_T(e, s_a, s) \hat{=} \begin{cases} \top, \text{ if } \nexists i: det(e, s_a, s_i) \\ \min\{t_i \mid \det(e, s_a, s_i)\}, \text{ otherwise} \end{cases}
$$

The publish/subscribe approach moves responsibility towards the environment: We consider a system in which the deferred choice smart contract subscribes to an external variable, and is notified of any changes with transactions. In these transactions, the smart contract could be sure that they have received a complete picture of all conditional event occurrences up to that point, and $\det_T(e, s_a, s_i)$ would be equal to $\det(e, s_a, s_i)$ until an event detection.

A core difference between the approaches is how quickly conditional events are detected after the occur. For the history approach, this entirely depends on the timing of transactions, as events can be detected retroactively. The publish/subscribe approach, conversely, ensures a very timely reaction to conditional events since we assume an active notification. In contrast to the other event types, though, both the approaches for conditional events currently lack support in practice, which we will examine in the next section.

5 EXTENDED ORACLE ARCHITECTURES

Oracles are needed to implement conditional events within smart contracts, since they rely on the values of external variables. In addition, these oracles must provide certain functionality to support the non-continual semantics—that is, they must either deliver historical data, or provide publish/subscribe services (see

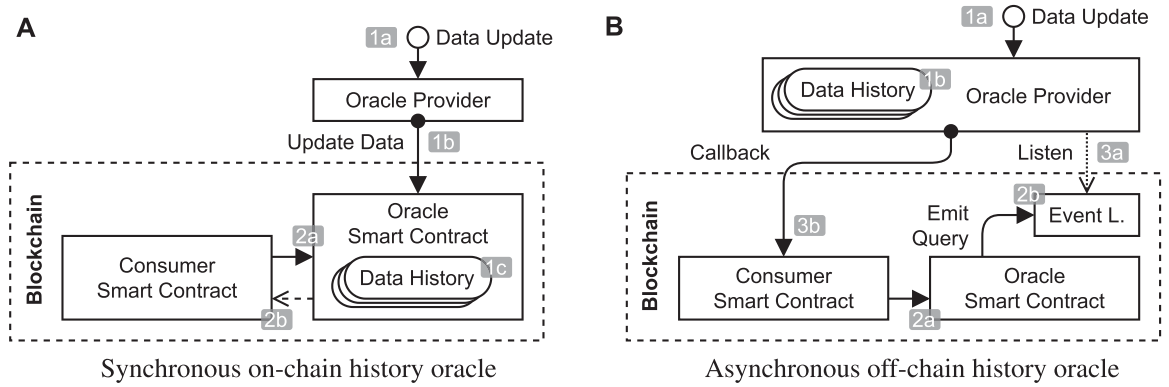

FIGURE 4 | Architecture and behavior of the history oracles.

Section 4.5). However, existing oracle architectures like storage and request/response are not capable of either.

In our previous work, we sketched purposeful extensions of existing oracle architectures precisely targeting these capabilities (Ladleif et al., 2020). In the following, we extend on the cursory textual descriptions in our previous work by providing concrete architecture and interface specifications tailored to the formal semantics introduced in **Section 4**.

5.1 History Oracles

History oracles allow consumers to obtain not only the present value of an external data variable, but also a history of past values. This makes it possible to implement the history approach at timed event detection explained in **Section 4.5**. We introduce two variants of the history oracle:

5.1.1 Architecture

The synchronous on-chain history oracle (see **Figure 4A**) is a direct extension of the existing storage oracle (see **Figure 3A**). The oracle provider observes the external data variable, and on data updates (1a) sends a transaction containing the new value to the oracle smart contract (1b). Instead of overwriting the old value, however, the new value is added to an on-chain database holding a full history of the data (1c). A consumer can then query a slice of this historical data (2a) and immediately receive result (2b).

The asynchronous off-chain history oracle (see **Figure 4B**) likewise extends the existing request/response oracle (see **Figure 3B**). Again, the oracle provider listens to updates of the external variable (1a), but stores a full history of the valuation off-chain (1b). A consumer smart contract may now submit a query for a slice of the data to the oracle smart contract (2a), which is emitted using the blockchain's event layer (2b). On receiving the query (3a), the oracle provider subsequently provides the requested data in a new transaction (3b).

5.1.2 Interfaces

Table 2 shows the interfaces of all oracle architectures described in this paper in terms of the formalization in **Section 4**. Regular storage and request/response oracles, for instance, do not require any input from the consumer smart contract, and return the current value of the external variable from D they target.

Regular history oracles are supplied one parameter, a timestamp from $\mathbb{N}$ at which to start the slice of historical values. As an output, consumers will get a list of timestamped data values. Specific points in time can then be found, for example, using a simple binary search.

5.2 Publish-Subscribe Oracles

As the name suggests, publish/subscribe oracles employ the well-known publish/subscribe software pattern, and let consumers subscribe to a specific external variable to actively notify them immediately of any changes. The effect is that no historical data needs to be stored, which corresponds to the publish/subscribe approach presented in **Section 4.5**. However, the onus of providing those updates in time is shifted to an off-chain component, in this case the oracle provider.

5.2.1 Architecture

Figure 5 shows the architecture of the publish/subscribe oracle. The off-chain oracle provider observes the external data variable, and on updates (1a) stores the most recent value locally (1b) and also provides it to all subscriber smart contracts already registered (1c). To subscribe, a consumer smart contract queries the oracle smart contract (2a) which emits an appropriate event (2b). This event is picked up by the oracle provider (3a), which adds the consumer smart contract to the list of subscribers (3b) and also provides the current value immediately to avoid any gaps (3c).

TABLE 2 | Interfaces of the oracles from the perspective of the consumer.

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	$\mathbb{D}$
Conditional	Storage, req/res	EXPR	{true, false}
	History	$\mathbb{N} \times$ EXPR	$\mathbb{N}$
	Publish/subscribe	EXPR	none

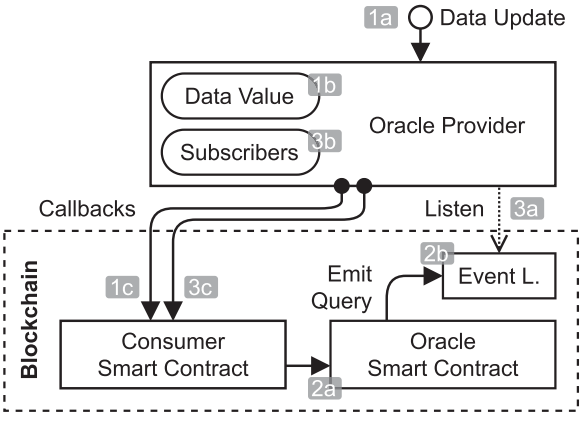

FIGURE 5 | Architecture and behavior of the publish-subscribe oracle.

5.2.2 Interfaces

From an interface perspective, the regular publish/subscribe oracle is equal to the traditional oracles (see **Table 2**). It does not require additional input, and provides a singular value each time an update occurs. Note that we do not consider unsubscription in detail in the scope of this paper. However, it could trivially be implemented by exposing an unsubscription function, which leads to the consumer smart contract being removed from the list of subscribers.

5.3 Conditional Oracle Variants

The oracle architectures above provide a strict separation of responsibilities when it comes to evaluating conditional events: the oracle provides the current value of the external variable, and the process smart contract evaluates specific conditions based on these values locally. This pattern is realistic in that it protects the business knowledge of an organization—conditions themselves may be confidential and stored in protected areas of the blockchain.

However, the cost of operating a system on a blockchain platform heavily depends on both the amount of transactions being performed as well as the size of their payload, owing to the limited amount of storage space available (Wood, 2014). Thus, we propose conditional oracles as a trade-off between confidentiality and cost by externalizing the evaluation of the condition attached to a conditional event to the oracle, either within its on-chain or off-chain infrastructure. The goal is to cut down on the number of necessary transactions and the amount of data being exchanged.

The conditional variants alter the interfaces of the oracles as shown in the lower half of **Table 2**. As an additional input, an expression from the domain EXPR of all possible expressions is given. We do not specify the structure of those expressions in detail, but require them to yield a Boolean result and only reference external variables the specific oracle provides.

The output will then reflect the evaluation of the condition on the basis of the values of the external variable. This is particularly interesting for history oracles, which either return the earliest timestamp within the bounds given by the start timestamp at which the condition was fulfilled, or $\top$ to express the condition never evaluated to *true*—essentially reproducing the detection function given in **Section 4.5**. The conditional publish/subscribe oracle, on the other hand, returns no specific value at all. Instead, a transaction is sent as a signal as soon as the condition becomes *true*. Overall, this reduces both the amount of transactions needed as well as the payload size of the remaining transactions.

6 PROTOTYPICAL IMPLEMENTATION

To prove the feasibility of the non-continual semantics and oracle architectures, we developed a novel prototypical implementation. It enables deployment of individual deferred choices containing message, timer, and conditional events on the blockchain, outside the context of their process. The source code with all necessary information to reproduce our results is available online.[1]

6.1 System Design

The design of the prototype is narrowly aligned to that of the various oracle architectures. There are three essential groups of components: 1) the blockchain smart contracts, 2) the off-chain oracle providers, and 3) an off-chain simulation framework. **Figure 6** shows the overall structure as an UML class diagram. The non-standard contract and event stereotypes are used for smart contract classes and event types, respectively.

The general design goal was to achieve a level playing field for comparing the different semantics and oracles. That is, we refrained from selective optimization of individual approaches. At the same time, the system design is laid out to be as generic as possible, for example streamlining the interfaces to a minimum as explained later in this section.

A major theme of the prototype is a strict separation between synchronous and asynchronous oracles, which results in the dual class layout for the oracle providers, oracle smart contracts, and deferred choices. For each concrete oracle architecture (see **Section 5**), e.g., the synchronous on-chain history oracle, there is a set of three corresponding concrete classes. All custom code related to single or a group of oracle architectures can be moved to these classes. In the following, we will walk through each part of the prototype.

6.2 Blockchain Smart Contracts

We use Ethereum (Wood, 2014) as our target blockchain, since it provides all features necessary and its development ecosystem is well-maintained and accessible. The core process logic is contained within smart contracts, 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 $\top$ is set to the type's largest value $2^{256} - 1$ which is "sufficiently" (see Definition 10) far—several hundred billion years—in the future.

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

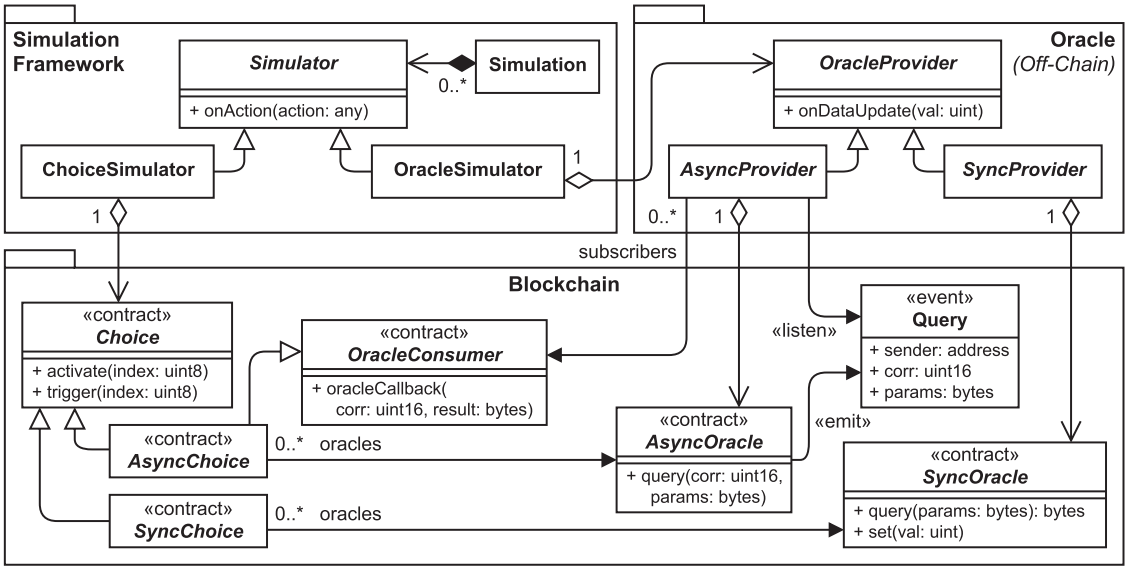

FIGURE 6 | Overall design of the prototype as a Unified Modeling Language (UML) class diagram.

6.2.1 Oracles

The smart contract classes AsyncOracle and SyncOracle contain the interfaces of the oracles (see **Figure 7**, **Section 6.2.2**). Parameters and query results are encoded in raw byte arrays (bytes) via the same mechanism Ethereum uses to encode transactions. This allows us to use common interfaces for all oracles, from which data can be extracted according to specific interfaces (see **Table 2**). Larger payloads will incur a higher cost following the rules in the standard (Wood, 2014).

For *synchronous* oracles, the current value or historical values are stored on the blockchain and updated/appended via set. Synchronous oracles may thus directly return a result upon query being called. For *asynchronous* oracles, the off-chain oracle provider stores 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 contract 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, Provable.[2] returns a unique query ID, which is attached alongside the query result for later matching. In our prototype, consumers choose an ID themselves in the form of the corr value.

[2]https://provable.xyz/

6.2.2 Deferred Choice

The contract class Choice and its children implement the state and behavior of deferred choice. Calling activate via a transaction initially activates the deferred choice (equivalent to picking a valid initial state from $\mathcal{CS}_0$), and trigger is used to perform a step of the transition relation $\overset{T}{\rightsquigarrow}$ and potentially trigger an event and pick it as the winner.

In this context, the non-determinism of the transition system is an issue—if multiple events are detected at the same time, both are valid winners and there is no sense of priority (see **Section 4.3**). Smart contracts are deterministic, though, and one event needs to be chosen. To this end, we opted for a two-phase strategy: Transactions may include a preferred event e_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 events' internal indices.

Figure 7 shows a schematic overview of the actions performed within all transactions depending on the oracle type. The complexity of the asynchronous oracles becomes especially apparent since the smart contract needs to keep track of all oracle callbacks, and asynchronously continue the event detection across transactions once the required external variable values are all gathered.

6.3 Off-Chain Components

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

6.3.1 Oracle Providers

The oracle providers are responsible for bridging the gap between the oracle smart contract and the external variable they observe. They manage communication responsibilities, mainly updating

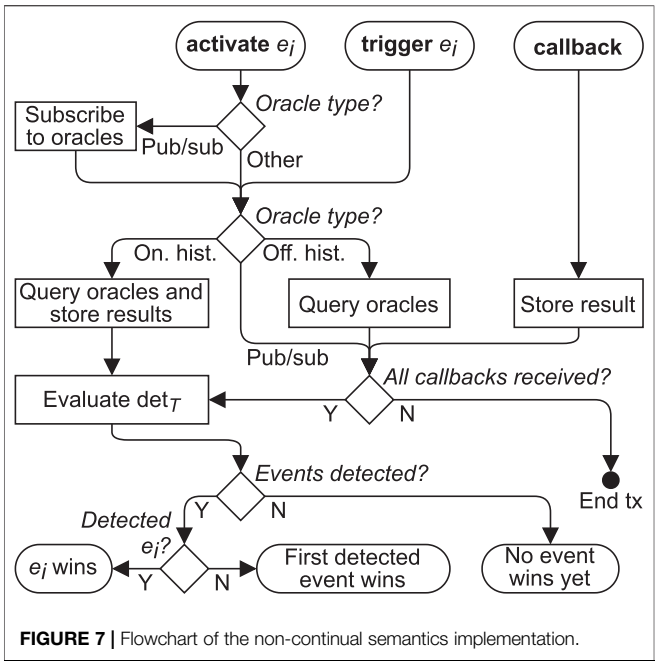

FIGURE 7 | Flowchart of the non-continual semantics implementation.

the oracle smart contract and sending responses to consumer requests depending on the oracle type. In the scope of this paper, we do not further investigate the connection to the original source of the variable. Existing approaches using RESTful APIs or similar can be used in practice.

6.3.2 Simulation Framework

To evaluate our proposals, we implemented a framework to simulate oracles and deferred choices in a reproducible way. For example, one such simulation may re-enact the scenario used in our running example in **Table 1**. To this end, a set of Simulator instances replay pre-defined lists of actions on their targets, e.g., sending data updates to an oracle provider or calling activation and trigger logic on a deferred choice contract.

7 EVALUATION

We evaluated our approach using the prototype from the three perspective mentioned at the beginning of this paper (see **Section 1**), namely correctness, cost, and immediacy. To this end, we performed a number of simulations on a private Ethereum 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 four CPUs. The detailed specification and raw results of all simulations are available online alongside the prototype.

7.1 Correctness

To verify the correctness of the non-continual semantics implementation, we generated n random deferred choices, each with k events $e_0, \ldots, e_{k-1}$. An associated simulation timeline was chosen specifically such that the events occur sequentially in order; that is, a transaction is sent for explicit events, a deadline is passed for timer events, or a condition on an oracle becomes true for conditional events. As a result, event e_0 is always the unambiguous winner and must be chosen by a correct approach.

TABLE 3 | Correctness of approaches in $n = 60$ random scenarios.

Semantics	Oracle	Correctness	
		Regular	Conditional
Continual	Storage	35%	35%
	Req/res	35%	35%
Non-continual	On-c. history	100%	100%
	Off-c. history	100%	100%
	Pub/sub	100%	100%

To serve as a baseline, we implemented the regular storage and request/response oracles and the continual semantics as well. Each of the 10 oracle variants was then used to simulate $n = 60$ scenarios, half of which with $k = 5$ events and the other half with $k = 10$ events. With a delay of 60 s between subsequent groups of transactions, the experiment took around 4 days (99:22:10) in total.

The share of simulations which yielded the correct winner e_0 is shown in **Table 3** for each regular and conditional oracle variant. The non-continual semantics approaches perform without fail, giving evidence that they indeed describe the intended behavior of deferred choice and that the implementation is accurate. As expected, the continual semantics using the traditional oracles encounter issues when certain event configurations resembling the problematic example in **Section 1** are generated—they only pick the correct winner in around 35% of cases, i.e., those in which the first event randomly turns out be explicit.

7.2 Cost

Cost is a major factor that influences the adoption of any 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 the cost in cryptocurrency. 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.

7.2.1 Simulation Design

All simulation scenarios follow the same pattern, in which c deferred choices consisting of exactly one conditional event each access a single shared oracle. The oracle receives u data updates. A trigger transaction is sent to each deferred choice at each fifth update, of which only the last will lead to the conditional event's detection 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.

TABLE 4 | Average smart contract deployment cost.

Oracle	Avg. deployment cost (10^3 gas)			
	Oracles		Deferred Choices	
	Regular	Conditional	Regular	Conditional
Storage		408/+48%	1,431/+3%	1,406/+1%
Req/res	281/+2%	281/+2%	1,502/+8%	1,477/+7%
On-c. hist	467/+69%	552/+100%	1,520/+10%	
Off-c. hist	281/+2%	281/+2%	1,592/+15%	1,448/+4%
Pub/sub	281/+2%	281/+2%	1,577/+14%	1,490/+7%

7.2.2 Deployment Cost

Initially, the smart contracts need to be created on the blockchain, incurring a one-time deployment cost contingent on the code size. **Table 4** shows the average deployment costs 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 thus more expensive to deploy.

For deferred choice, 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 choice itself, and that the more powerful approaches like publish/subscribe and history oracles require larger smart contracts.

7.2.3 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 **Figure 8**.

Several observations are immediately apparent: All synchronous oracles (**Figures 8A–D**) become relatively less expensive the more consumers share the cost, as visible by the decline along the y-axis. This is not the case for asynchronous oracles (**Figures 8E–J**), as their cost linearly scales with the number of consumers.

Naturally, 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 (**Figures 8C,D**) and the regular off-chain history (**Figure 8G**) oracles, which show an exponential trajectory on the x-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.

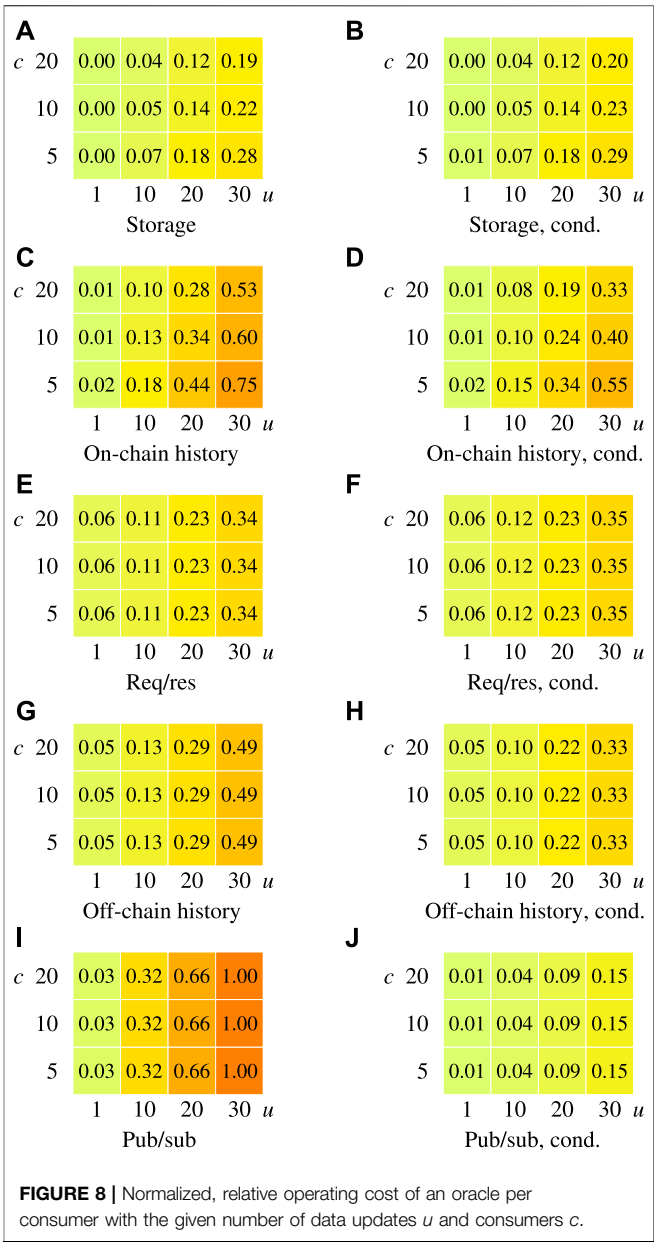

FIGURE 8 | Normalized, relative operating cost of an oracle per consumer with the given number of data updates u and consumers c.

8 DISCUSSION AND CONCLUSION

In this paper, we have introduced several formal as well as practical solutions in the form of novel oracle patterns to support deferred choice within smart contracts on blockchain

TABLE 5 | Relative comparison of the proposed oracle architectures.

Variant	Oracle pattern	Correctness	Cost	Immediacy
Regular	On-chain history	✓	Exponential (storage)	Arbitrary
	Off-chain history	✓	Exponential (transaction)	Arbitrary
	Publish/subscribe	✓	Linear	Immediate
Conditional	On-chain history	✓	Exponential (storage)	Arbitrary
	Off-chain history	✓	Linear	Arbitrary
	Publish/subscribe	✓	Linear/fixed	Immediate

networks. In the following, we will compare these solutions, discuss their shortcomings and limitations, and conclude with a summary of our contributions.

8.1 Comparison and Assessment

In **Section.1**, our central research question gave rise to three metrics suitable to judge our proposed solution: correctness, cost, and immediacy. **Table 5** shows an overview of how the oracle patterns perform in these metrics.

The first metric, correctness, is readily achieved by all of our proposed solutions by design as evidenced by the underlying formal framework (see **Section 4**). The experimental evaluation (see **Section 7.1**) supports this impression by showing that our proposed solutions can effectively choose the correct winner of a deferred choice despite the restrictions posed by the transaction-driven blockchain environment. Supporting deferred choice comes at a price, however, and the required novel oracle architectures tend to be more expensive than existing approaches with less functionality both during deployment and runtime (see **Section 7.2**). More gravely, due to the platform's storage limitations, some approaches might not be feasible for all data sources and scenarios depending on the number of data updates: We have shown that on-chain history oracles might quickly become infeasible when the exponential pricing of storage in Ethereum fully comes into play. The regular off-chain history oracle similarly experiences an exponential growth in cost due to the rising size of transaction payloads. There are, of course, conceivable solutions to this, e.g., only storing and submitting data for a certain amount of time to keep demands on a constant level.

The other proposed solutions exhibit a linear rise in cost the more data updates there are, albeit on different baselines. The regular publish/subscribe oracle shows the highest cost overall, while the conditional variant shows the lowest. The latter conditional publish/subscribe oracle even has the potential for a fixed cost if unsubscription mechanisms were to be implemented. In conclusion, we believe that practical implementations in usable blockchain-based process engines will be necessary to really show the feasibility of supporting deferred choice patterns in practice, for which we have laid the groundwork. Lastly, the experimental evaluation is based on procedurally generated scenarios with a known outcome. Thus, the important aspect of immediacy—i.e., how quickly a deferred choice is resolved after an event occurs—is not directly covered by our experimental evaluation, since it is a direct result of the scenarios' structure. Still, it is evident that the publish/subscribe approaches will achieve a high level of immediacy in practice by design: Whenever the deferred choice state could potentially evolve, a transaction is automatically sent within a very short timespan. The other approaches, though, will always be hindered by the need for a manually or periodically submitted transaction to arrive and trigger the evaluation of the deferred choice. This ultimately appears to be a fundamental restriction of any solution to deferred choice within smart contracts or other transaction-based environments.

8.2 Limitations and Future Work

The present work is subject to several limitations. Perhaps most importantly, we did not take into account some concrete network and protocol delays that are present in blockchains and Ethereum in particular: There is no notion of forks, side chains, or confirmation time. The semantics and implementation also do not consider the possibility of blockchain exceptions, like running out of gas, which may lead to the premature termination of transactions. As such, our paper presents results obtained in an ideal environment. At the same time, this point of view helps keeping our results generalizable for other Ethereum-like blockchains in turn. We did not take into account blockchains which are fundamentally different than Ethereum, and the topic of deferred choice using such other blockchain platforms surely deserves attention in the future. Further, the prototype is not production-ready. Authentication and security features were omitted completely, and several optimizations like specialized interfaces for individual oracle architectures or low-level optimizations using Solidity Assembly were left out to not unfairly influence the direct comparison of the oracles. We also do not support some deferred choice and oracle configurations, e.g., multiple types of oracles being used in the same deferred choice instance. These design decisions were necessary to keep the development scope manageable. Lastly, we focused on rather direct and pragmatic solutions to the issue of deferred choice within smart contracts. The proposals of course come with tradeoffs regarding the fundamental properties of blockchain technology. For instance, as mentioned in **Section 5.3**, the conditional oracle variants externalize the evaluation of conditions attached to conditional events to a third-party oracle provider. This may not always be desirable or even possible in practice. At the same time, moving such responsibility off-chain may also impact the fundamental properties of blockchain technology itself: If the oracle performs calculations, they can not immediately be validated by the blockchain network. A study on the consequences for auditing and non-repudiation would be

necessary before releasing such solutions into production environments.

8.3 Summary of Contributions

To conclude, providing full support for deferred choice patterns within smart contracts on contemporary blockchain technology is a complex issue—especially when considering heterogenous scenarios with multiple involved event types and external data sources. Guided by our initial research question (see **Section 1**), we suggested a set of solutions which we formally described, prototypically implemented, and experimentally evaluated. In summary, our contributions are as follows:

- We provided a tailor-made formal semantics of deferred choice with heterogenous event types in non-continual environments, taking into account the role of the external environment in event occurrences. The semantics offer a formal footing which allows the reasoning about and alignment of our implementation and evaluation.
- We introduced two approaches (history and publish/subscribe) realizing this semantics within smart contracts on blockchain networks, i.e., taking into account the isolation and non-continuity restrictions of blockchains. The approaches were formally introduced, and form the basis for the introduction of concrete oracle architectures to support them in practice.
- Lastly, we implemented all oracle architectures and variants introduced in this paper using the Ethereum blockchain (Wood, 2014). While the main purpose of the implementation was to experimentally evaluate certain properties of our solutions, it may also serve as a blueprint for process engines in the future.

Overall, we hope our contributions serve to improve support for deferred choice patterns within smart contracts on blockchain, something largely absent in state-of-the-art blockchain-based process execution approaches.

DATA AVAILABILITY STATEMENT

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found below: https://github.com/bptlab/blockchain-deferred-choice.

AUTHOR CONTRIBUTIONS

All authors contributed to the conception and steering of the topic. JL wrote the article, implemented the prototypical implementation, and performed any associated experiments. All authors contributed to article revision, read, and approved the submitted version.

ACKNOWLEDGMENTS

A previous version of this article was published online as a preprint (Ladleif and Weske, 2021).

REFERENCES

Abid, A., Cheikhrouhou, S., and Jmaiel, M. (2020). "Modelling and Executing Time-Aware Processes in Trustless Blockchain Environment," in *Risks and Security of Internet and Systems, CRiSIS 2019 of Lecture Notes in Computer Science*. Editors S. Kallel, F. Cuppens, N. Cuppens-Boulahia, and A Hadj Kacem (Cham: Springer), 12026, 325–341. doi:10.1007/978-3-030-41568-6_21

Adams, M., Suriadi, S., Kumar, A., and ter Hofstede, A. H. M. (2020). "Flexible Integration of Blockchain with Business Process Automation: A Federated Architecture," in *Advanced Information Systems Engineering of Lecture Notes in Business Information Processing*. Editors N. Herbaut and M. La Rosa (Cham: Springer), 386, 1–13. doi:10.1007/978-3-030-58135-0_1

Al-Breiki, H., Rehman, M. H. U., Salah, K., and Svetinovic, D. (2020). Trustworthy Blockchain Oracles: Review, Comparison, and Open Research Challenges. *IEEE Access* 8, 85675–85685. doi:10.1109/ACCESS.2020.2992698

Azzopardi, S., Ellul, J., and Pace, G. (2021). "Runtime Monitoring Processes Across Blockchains," in 9th IPM International Conference on Fundamentals of Software Engineering 2021 (FSEN 2021), Tehran, Iran, May 19-21, 2021.

Cheikhrouhou, S., Kallel, S., Guermouche, N., and Jmaiel, M. (2015). The Temporal Perspective in Business Process Modeling: A Survey and Research Challenges. *Serv. Oriented Comput. Appl.* 9, 75–85. doi:10.1007/s11761-014-0170-x

Corradini, F., Fornari, F., Polini, A., Re, B., and Tiezzi, F. (2018). A Formal Approach to Modeling and Verification of Business Process Collaborations. *Sci. Comput. Programming* 166, 35–70. doi:10.1016/j.scico.2018.05.008

Corradini, F., Marcelletti, A., Morichetta, A., Polini, A., Re, B., and Tiezzi, F. (2020). "Engineering Trustable Choreography/Based Systems Using Blockchain," in SAC '20: Proceedings of the 35th Annual ACM Symposium on Applied Computing, Brno, Czech Republic, March-April 30-3, 2020 (New York, NY, USA: Association for Computing Machinery), 1470–1479. doi:10.1145/3341105.3373988

Dijkman, R. M., Dumas, M., and Ouyang, C. (2008). Semantics and Analysis of Business Process Models in Bpmn. *Inf. Softw. Technol.* 50, 1281–1294. doi:10.1016/j.infsof.2008.02.006

Eder, J., Panagos, E., and Rabinovich, M. (1999). "Time Constraints in Workflow Systems," in *Advanced Information Systems Engineering of Lecture Notes in Computer Science*. Editors M. Jarke and A. Oberweis (Berlin, Heidelberg: Springer), 1626, 286–300. doi:10.1007/3-540-48738-7_22

García-Bañuelos, L., Ponomarev, A., Dumas, M., and Weber, I. (2017). "Optimized Execution of Business Processes on Blockchain," in Intl. Conference on Business Process Management, Barcelona, Spain, September 10-15, 2017 (Springer), 130–146. doi:10.1007/978-3-319-65000-5_8

Garcia-Garcia, J. A., Sánchez-Gómez, N., Lizcano, D., Escalona, M. J., and Wojdynski, T. (2020). Using Blockchain to Improve Collaborative Business Process Management: Systematic Literature Review. *IEEE Access* 8, 142312–142336. doi:10.1109/access.2020.3013911

Houhou, S., Baarir, S., Poizat, P., and Quéinnec, P. (2019). "A First-Order Logic Semantics for Communication-Parametric Bpmn Collaborations," in *Business Process Management*. Editors T. Hildebrandt, B. van Dongen, M. Röglinger, and J. Mendling (Cham: Springer International Publishing), 52–68. doi:10.1007/978-3-030-26619-6_6

Kheldoun, A., Barkaoui, K., and Ioualalen, M. (2017). Formal Verification of Complex Business Processes Based on High-Level Petri Nets. *Inf. Sci.* 385-386, 39–54. doi:10.1016/j.ins.2016.12.044

Klinger, P., and Bodendorf, F. (2020). "Blockchain/Based Cross/Organizational Execution Framework for Dynamic Integration of Process Collaborations," in 15th International Conference on Wirtschaftsinformatik (WI), Potsdam, Germany, March 8-11, 2020 893–908. doi:10.30844/wi_2020_i2-klinger

Kossak, F., Illibauer, C., and Geist, V. (2012). "Event-Based Gateways: Open Questions and Inconsistencies," in *Business Process Model and Notation*. Editors J. Mendling and M. Weidlich (Berlin, Heidelberg: Springer Berlin Heidelberg), 53–67. doi:10.1007/978-3-642-33155-8_5

Ladleif, J., Weber, I., and Weske, M. (2020). "External Data Monitoring Using Oracles in Blockchain-Based Process Execution," in *Business Process Management: Blockchain and Robotic Process Automation Forum, BPM 2020 of Lecture Notes in Business Information Processing*. Editors A. Asatiani, J M García, N Helander, and A Jiménez-Ramírez (Cham: Springer), 393, 67–81. doi:10.1007/978-3-030-58779-6_5

Ladleif, J., and Weske, M. (2020). "Time in Blockchain-Based Process Execution," in 24th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2020, Eindhoven, The Netherlands, October 5-8, 2020 (IEEE), 217–226. doi:10.1109/EDOC49727.2020.00034

Ladleif, J., Weske, M., and Weber, I. (2019). "Modeling and Enforcing Blockchain-Based Choreographies," in *Business Process Management. BPM 2019 of Lecture Notes in Computer Science*. Editors T. Hildebrandt, B. van Dongen, M. Röglinger, and J. Mendling (Cham: Springer), 11675, 69–85. doi:10.1007/978-3-030-26619-6_7

Ladleif, J., and Weske, M. (2021). Which Event Happened First? Deferred Choice on Blockchain Using Oracles. CoRR abs/2104.10520

López-Pintado, O., Dumas, M., García-Bañuelos, L., and Weber, I. (2019a). "Interpreted Execution of Business Process Models on Blockchain," in 2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC), Paris, France, October 28-31, 2019 206–215. doi:10.1109/EDOC.2019.00033

López-Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I., and Ponomarev, A. (2019b). Caterpillar: A Business Process Execution Engine on the Ethereum Blockchain. *Softw. Pract. Exper* 49, 1162–1193. doi:10.1002/spe.2702

Lu, Q., Binh Tran, A., Weber, I., et al. (2021). Integrated Model-Driven Engineering of Blockchain Applications for Business Processes and Asset Management. *Softw. Pract. Exper*. 51, 1059–1079. doi:10.1002/spe.2931

Madsen, M. F., Gaub, M., Høgnason, T., Kirkbro, M. E., Slaats, T., and Debois, S. (2018). "Collaboration Among Adversaries: Distributed Workflow Execution on a Blockchain," in Symposium on Foundations and Applications of Blockchain, Los Angeles, California, USA, March 9, 2018.

Mendling, J., Weber, I., Aalst, W. V. D., Brocke, J. V., Cabanillas, C., Daniel, F., et al. (2018). Blockchains for Business Process Management - Challenges and Opportunities. *ACM Trans. Manage. Inf. Syst*. 9, 1–16. doi:10.1145/3183367

Nakamoto, S. (2008). *Bitcoin: A Peer-To-Peer Electronic Cash System*.

OMG (2013). *Business Process Model and Notation (BPMN)*. Version 2.0.2. Object Management Group (OMG).

Plotkin, G. D. (1981). *A Structural Approach to Operational Semantics*. Aarhus, Denmark: Computer Science Department, Aarhus University Denmark.

Russell, N., Ter Hofstede, A. H., Van Der Aalst, W. M., and Mulyar, N. (2006). Workflow Control-Flow Patterns: A Revised View. BPM Center Report BPM-06-22, BPMcenter. org 06–22

Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A., and Mendling, J. (2016). "Untrusted Business Process Monitoring and Execution Using Blockchain," in *Business Process Management (BPM) of Lecture Notes in Computer Science*. Editors M. L. Rosa, P. Loos, and O. Pastor (Cham: Springer), 9850, 329–347. doi:10.1007/978-3-319-45348-4_19

Weske, M. (2019). *Business Process Management*. 3rd edn. Berlin, Heidelberg: Springer. doi:10.1007/978-3-662-59432-2

Wood, G. (2014). Ethereum: A Secure Decentralised Generalised Transaction Ledger. Tech. rep. Ethereum Project Yellow Paper

Xu, X., Pautasso, C., Zhu, L., Lu, Q., and Weber, I. (2018). "A Pattern Collection for Blockchain-Based Applications," in 23rd European Conference on Pattern Languages of Programs (EuroPLoP) (ACM), Irsee, Germany, July 4-8, 2018. doi:10.1145/3282308.3282312

Xu, X., Weber, I., Staples, M., Zhu, L., Bosch, J., Bass, L., Pautasso, C., and Rimba, P. (2017). "A Taxonomy of Blockchain-Based Systems for Architecture Design," in IEEE Intl. Conf. Software Architecture (ICSA), Gothenburg, Sweden, April 3-7, 2017 243–252. doi:10.1109/ICSA.2017.33

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.