Expert Systems With Applications 184 (2021) 115578

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

CEPchain: A graphical model-driven solution for integrating complex event processing and blockchain

Juan Boubeta-Puig [a,*], Jesús Rosa-Bilbao [b], Jan Mendling [c]

[a] UCASE Software Engineering Research Group, Department of Computer Science and Engineering, University of Cadiz, Avda. de la Universidad de Cádiz 10, 11519 Puerto Real, Cádiz, Spain

[b] UCASE Software Engineering Research Group, School of Engineering, University of Cadiz, Avda. de la Universidad de Cádiz 10, 11519 Puerto Real, Cádiz, Spain

[c] Institute for Information Business, Vienna University of Economics and Business, Welthandelsplatz 1, 1020 Vienna, Austria

ARTICLE INFO

Keywords:
Complex event processing
Blockchain
Smart contract
Model-driven development
Graphical modeling tool
Supply chain

ABSTRACT

Blockchain provides an immutable distributed ledger for storing transactions. One of the challenges of blockchain is the particular processing of dynamic queries due to accumulating costs. Complex Event Processing (CEP) provides efficient and effective support for this in a way, however, that is difficult to integrate with blockchain. This paper addresses the research challenges of integrating blockchain with CEP. More specifically, we envision an effective development environment in which (i) event-driven smart contracts are modeled in a graphical way, which are, in turn, (ii) automatically transformed into complementary code that is deployed in both a CEP engine and a blockchain network, and then (iii) executed on off-chain CEP applications which, connected to different data sources and sinks, automatically invoke smart contracts when event pattern conditions are met. We follow a classic systems engineering approach for defining the concepts of our system, called CEPchain, which addresses the described requirements. CEPchain was evaluated using a real-world case study for vaccine delivery, which requires an unbroken cold chain. The results demonstrate that our approach can be applied without requiring experts on event processing and smart contract languages. Our contribution simplifies the design of integrated CEP and blockchain functionality by hiding implementation details and supporting efficient deployment.

1. Introduction

Blockchain is an emerging technology that supports the exchange of tangible and intangible assets without the need for third-party intermediaries (Preukschat, 2017). Gartner estimates that by 2023, blockchain will support the global movement and tracking of $2 trillion/year of goods and services (Gartner, 2020). Conceptually, blockchain is based on a distributed digital ledger of cryptographically signed transactions, which are grouped into blocks. Each block is cryptographically linked to the previous one once validated (Yaga et al., 2018). A new block is replicated across the blockchain network in such a way that conflicts are automatically resolved. In this way, blockchain addresses functional requirements of data storage, communication services and computation services, as well as non-functional requirements of immutability, transparency, integrity, non-repudiation and equal rights (Xu et al., 2019).

The behavior of the blockchain can be programmed using smart contracts. Smart contracts can be used to specify agreements between different parties at design time and to validate the fulfillment of what was agreed upon conditions at runtime. However, the implementation of smart contracts is a cumbersome and difficult task not only for domain experts, but also for software developers for several reasons. Firstly, programming requires advanced knowledge of special-purpose languages such as Solidity, which is executable on the Ethereum Virtual Machine (EVM) (Ethereum Foundation, 2021). Secondly, languages like Solidity do not directly support business rules with temporal event correlation and have constraints on the value types and length. For instance, long-type and fixed-point numbers are not supported by the current 0.6.8 Solidity version (Ethereum, 2021), which implies limitations in defining more complex and temporal logic for smart contracts. Thirdly, on permissionless blockchain platforms such as Ethereum, smart contracts are executed locally by miners, which are compensated for validating and recording transactions. The required data computation and storage to conduct each transaction, data storage, function execution and contract deployment can be prohibitively high for scenarios that require continuous querying.

* Corresponding author.
E-mail addresses: juan.boubeta@uca.es (J. Boubeta-Puig), jesus.rosabilbao@alum.uca.es (J. Rosa-Bilbao), jan.mendling@wu.ac.at (J. Mendling).

https://doi.org/10.1016/j.eswa.2021.115578
Received 17 August 2020; Received in revised form 26 April 2021; Accepted 6 July 2021
Available online 13 July 2021

In this paper, we address these research challenges. We build on the idea proposed by Fournier and Skarbovsky (2019) of enriching smart contracts with temporal business rule processing using Complex Event Processing (CEP) (Luckham, 2012). In this way, temporal queries can be delegated from the blockchain to a CEP component that provides temporal reasoning in real time by matching event patterns over time and quickly reacting to them. More specifically, we address the following concrete requirements, which have not been addressed by prior research:

- **R1**: Graphically modeling event-driven smart contracts.
- **R2**: Automatically transforming event-driven smart contract models into code and then deploying this code in both a CEP engine and a blockchain network.
- **R3**: Executing off-chain CEP applications which, connected to different data sources and sinks, automatically invoke smart contracts when event pattern conditions are met.

To this end, we build on concepts of Model-Driven Development (MDD) (Brambilla et al., 2017) to integrate CEP with blockchain via a graphical model-driven solution, called CEPchain. Our model-driven approach is subdivided into the three following layers. The off-chain design time layer supports the definition of high-level models of event-driven smart contracts, which can be transformed into code that is executable on a CEP engine and a blockchain network. The off-chain runtime layer is responsible for deploying all automatically generated code. The on-chain runtime layer provides the blockchain network in which a smart contract can be deployed and automatically invoked upon pattern detection. In order to avoid cost, the domain expert can decide which complex event types will invoke smart contract functions and which will not. In this way, complex event types that are intermediate can be managed and stored off the blockchain. Our approach was evaluated in a real-world case study for vaccine delivery, which requires an uninterrupted cold chain.

The rest of the paper is organized as follows. Section 2 describes the background of blockchain and CEP, and then relates our work to prior research. Section 3 presents CEPchain, our graphical model-driven proposal for integrating such technologies. Section 4 describes and discusses the application of the proposal to a real-world case study. Finally, Section 5 draws conclusions and identifies directions for future research.

2. Preliminaries

In this section, we describe the background of the blockchain and CEP technologies and discuss related works.

2.1. Blockchain

The concept known as blockchain is formed by several components. We follow the terminology used in Xu et al. (2019) and Yaga et al. (2018), and discuss these components in turn, namely blockchain, blockchain network, and smart contracts.

Firstly, a *blockchain* can be defined as a distributed digital ledger of cryptographically signed transactions, which are grouped into blocks. Each block is cryptographically linked to the previous one once it is validated and has undergone a consensus decision (Yaga et al., 2018). Secondly, a new block is replicated across copies of the ledger within the *blockchain network*, with any conflicts that occur being automatically resolved. Blockchain networks can be classified according to their permission model (Yaga et al., 2018): (i) *permissionless*, if anyone can publish a new block without requiring permission from any authority, (ii) *permissioned*, if only particular users, who are authorized by a centralized or decentralized authority, can publish blocks, and (iii) *consortium*, if a permissioned blockchain network is deployed and governed by a group of individuals and organizations. Ethereum (Ethereum Foundation, 2021) is an example of a permissionless blockchain platform, while Hyperledger Fabric (Linux Foundation, 2021) is a permissioned/consortium one.

Thirdly, *smart contracts* (aka. *chaincode* in some blockchain platforms such as Hyperledger Fabric) are programs that can be deployed and executed on a blockchain. Once deployed, the code of the smart contract is immutable and deterministic. Smart contracts can implement business logic, conditions and triggers to manage programmable transactions (Xu et al., 2019). A smart contract is a piece of executable code (*functions*), a *private storage* to register its internal state and, optionally, the *account balance* for cryptocurrency blockchain platforms such as Ethereum. On the Ethereum blockchain, smart contracts are implemented in the Solidity language and then compiled into bytecode, which is executed by the EVM. A *contract creation transaction* is needed to deploy a smart contract on the blockchain. The implementation code of the smart contract is contained in the transaction's payload. In order to create the smart contract on the blockchain, this transaction must be authorized by the sender's signature. Once the contract creation transaction is processed, a *contract address* will uniquely identify this smart contract. Upon smart contract deployment on blockchain, users of the Ethereum blockchain can send Ethers (ETHs) to this contract through a *monetary transaction*. Smart contracts can be invoked externally or from other smart contracts. By sending *contract invoking transactions* to a smart contract's address, users of the blockchain can invoke its functions. When an invoking transaction is conducted, a smart contract can emit events and store event logs in the blockchain. These event logs can be further consumed by external systems for data and process analytics (Mühlberger et al., 2019).

There are two ways of storing data in Ethereum smart contracts: (i) as a variable in a smart contract and (ii) as a log event. On the Ethereum blockchain, smart contracts are executed locally by *miners*, which are compensated for validating and recording transactions. The Ethereum yellow paper (Wood, 2019) proposes the cost model for quantifying the required data computation and storage to conduct each transaction, data storage, function execution and contract deployment. This cost is expressed in *gas*, i.e. the fee required to successfully execute a smart contract or conduct a transaction on Ethereum. Depending on the *gas price* established by the user, gas cost is converted to ETH, as explained below. According to this cost model, every transaction has a fixed cost of 21,000 gas. As an example, we can use the exchange rate of US$188/ETH from May 10th, 2020. We also assume a gas price of 2 x 10^{-9} ETH (2 Gwei) on Ethereum. The cost of storing data as a variable in a smart contract depends on the number of SSTORE operations required for the variable (Xu et al., 2019). Every simple type variable in Solidity (32-bytes) requires only one SSTORE operation that costs 20,000 gas plus 68 gas per every non-zero byte of data. So, the total cost of storing 32-byte data as a variable is 21,000 gas (transaction) + 20,000 gas (SSTORE operation) + 32 bytes x 68 gas = 43,176 gas x 2 x 10^{-9} x $188 = US$0.016.

Storing data as a log event requires a log topic that costs 375 gas plus 8 gas per every byte of data in the log topic. So, the total cost of storing 32-byte data as a log event is 21,000 gas (transaction) + 375 gas (topic) + 32 bytes x 8 gas = 21,631 gas x 2 x 10^{-9} x $188 = US$0.008. Therefore, storing data as log events is cheaper than as variables and can be consumed by other external systems for real-time data analysis. However, log events only allow up to three parameters to be queried. In contrast, storing data as a variable is more efficient to manage (Xu et al., 2019) but is less flexible because of the Solidity constraints on the value types and length; for instance, the long type and fixed point numbers are not supported by the current 0.6.8 Solidity version (Ethereum, 2021).

2.2. Complex event processing

CEP is a cutting-edge technology for real-time big data analytics (Boubeta-Puig et al., 2015a). CEP allows us to analyze and correlate

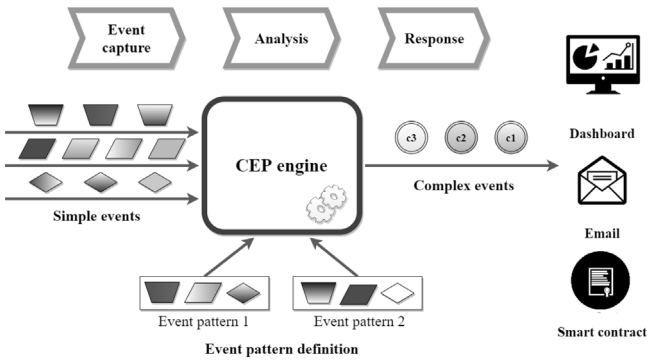

Fig. 1. CEP stages.

continuous streams of data on the fly by matching *event patterns* over time and quickly reacting to them. This technology fundamentally performs as a real-time expert system (Cummins, 2009).

An event occurrence or event sequence requiring an immediate reaction is known as a *situation* (Etzion & Niblett, 2010). While *simple events* are indivisible and occur at a specific point in time, *complex events* provide meaningful and valuable information by combining and summarizing other events that happen before (Event Processing Technical Society, 2011; Luckham, 2012). Complex events are automatically created by a *CEP engine* when the conditions previously defined in an event pattern are satisfied. A CEP engine is a powerful software that provides programmers with the ability to implement such patterns by using *Event Processing Languages* (EPLs) (Boubeta-Puig et al., 2014). However, the implementation of event patterns is generally a complex and cumbersome task (Burgueño et al., 2018). Additionally, a CEP engine can trigger real-time actions and alerts upon pattern detection.

Fig. 1 shows the three stages of CEP technology:

1. **Event capture**: the reception of real-time simple events to be analyzed and correlated. A simple event might be a new temperature sensor value.
2. **Analysis**: the prompt detection of situations of interest when the conditions defined in event patterns are met. As an example, in the context of a logistics scenario, the *TemperatureWarning* event pattern could detect when a temperature sensor value is outside the 2–8 °C range, warning that the current temperature of a vaccine is not the recommended by the World Health Organization (2006).
3. **Response**: the actions to be carried out for such detected situations and their notification to the interested consumers. For instance, if a *TemperatureWarning* is triggered, it may then be delivered directly to a monitoring dashboard, an email account or other applications. The novelty of the present paper is that the functions of a smart contract, already deployed in a blockchain, can be automatically invoked by complex events, such as the *TemperatureWarning* ones.

CEP technology has several benefits, such as faster and automatic reply, human workload reduction, information overload prevention and decision quality improvement (Chandy & Schulte, 2010). Since situations of interest can be detected and reported in real time, latency in the decision-making process is decreased in comparison to traditional event analysis techniques. Thereby, CEP-based systems have gained many applications in a variety of areas including logistics, monitoring critical infrastructure and financial applications (Hinze et al., 2009).

In this work, we propose the integration of CEP and blockchain to provide an external CEP-based system that can process the event-driven temporal logic outside smart contracts for the following two reasons. On the one hand, making inferences and reasoning on-chain can be costly, with it being more appropriate for heavy computation (inferences) to occur off-chain, thus reducing the cost of the associated transactions (Idelberger et al., 2016). For this reason, an off-chain CEP-based system is essential to make inferences and reasoning by matching event patterns and enriching the generated complex events with the ability to invoke smart contract functions upon pattern detection. This makes it possible to simplify the logic and size of smart contracts. As explained in Section 2.1, the smaller the size of a contract, the lower is its cost. On the other hand, smart contracts use business rules for defining conditions in which transactions occur, but lack the capability to reason over time (Fournier & Skarbovsky, 2019). In particular, smart contract languages support neither data windows nor temporal operators, and certain languages, such as Solidity, do not support long type and fixed point numbers, among others. Using EPLs supported by CEP engines, rich pattern expressions over events and time can be defined outside smart contracts.

Table 1
A comparison between our proposal and existing related works.

Approach	R1	R2	R3
Fournier's proposal (Fournier & Skarbovsky, 2019)	–	–	+/–
CEPchain (our proposal)	+	+	+

2.3. Related work

This subsection presents related work on model-driven approaches integrating CEP and blockchain with reference to the requirements established in Section 1, comparing our proposal with other existing ones. To the best of our knowledge, the work by Fournier and Skarbovsky (2019) is the only existing study that proposes a model-driven approach combining CEP and blockchain. Indeed, both works make use of the CEP technology to enable smart contracts to reason over time. However, as summarized in Table 1, there are significant differences between our proposal and that of Fournier.

Firstly, our proposal provides domain experts with a *graphical* modeling editor for designing event-driven smart contracts, i.e. smart contract functions that can be automatically invoked by complex events. The modeled pattern conditions are then automatically transformed into Esper EPL code, which is executable on an Esper CEP engine (EsperTech, 2021), and the smart contracts are transformed into Java code, which can interact with an Ethereum blockchain network (Ethereum Foundation, 2021). Therefore, while our proposal supports automatic model-to-text transformations, Fournier and Skarbovsky provide a textual editor, and code is generated manually. In addition, Fournier's proposal partly addresses requirement R3, providing a CEP application that is capable of managing smart contracts with temporal aspects, a simulator that generates temperature sensor readings and a mobile user interface that interacts with the system. Additionally, our proposal allows for the configuration and connection with different data sources and sinks such as files, message brokers and other external systems according to application domains needs.

Secondly, our proposal currently uses the Esper CEP engine and the Ethereum blockchain, while Fournier's makes use of the Proton CEP engine (Skarbovsky, 2020) and the Hyperledger Fabric blockchain (Linux Foundation, 2021). Since our solution is based on a graphical model-driven approach, new model-to-text transformation rules could be created and added for automatically transforming graphical pattern models into both code executable by other CEP engines such as Proton and Siddhi (WSO2, 2020), and code executable by other blockchain platforms, like Hyperledger Fabric. Depending on the particular needs of each real-world application domain, with examples being e-voting (Kshetri & Voas, 2018; Yang et al., 2020), health (Calvo et al., 2019; Kshetri, 2018), IoT security and fraud detection (Ali et al., 2019; Farrugia et al., 2020; Moin et al., 2019; Roldán et al., 2020), air pollution and road traffic (Díaz et al., 2020), business process outsourcing (Eshuis et al., 2016) and agricultural supply chains (Xu et al., 2019), the use of a specific blockchain platform could be more appropriate, as shown in Table 2.

Table 2
A comparison between the Ethereum and Hyperledger Fabric platforms.

Feature	Ethereum	Hyperledger Fabric
Purpose	Enterprises and generalized applications	Enterprises
Confidentiality	Transparent	Confidential transactions
Mode of participation	Public/private network and without permissions	Private and authorized network
Algorithm	Proof of work and proof of stake	Consensus (no mining required)
Decentralized	Yes	Yes
Cryptocurrency	Yes (ETH)	No
Need for blockchain knowledge	Yes	Yes
Automatic generation of smart contract code	No	No

3. The CEPchain approach to integration of blockchain and CEP

This section presents our novel model-driven approach to address the integration of CEP and blockchain. Recall that blockchain mainly provides immutability and transparency, but not data analysis in real time. This is why the integration of blockchain and CEP is adequate for application domains which require detecting critical situations in real time and storing them in a blockchain for transparency and immutability purposes.

Fig. 2 gives an overview of the CEPchain approach and its three layers: off-chain design time, off-chain runtime and on-chain runtime. The two off-chain layers consist of a set of software components that are kept off-chain and computed at design time (CEPchain graphical tool) or runtime (data sources, data sinks and the CEPchain graphical tool including a CEP engine). The third on-chain layer includes the components that are placed and computed on-chain (smart contract, account balance, private storage and event logs). These layers are deeply detailed in the following subsections.

The CEPchain approach addresses the three requirements established in Section 1. More specifically, R1 (graphically modeling event-driven smart contracts) is fulfilled by the off-chain design time layer, R2 (automatically transforming event-driven smart contracts models into code and then deploying this code in both a CEP engine and a blockchain network) is fulfilled by the off-chain design time and off-chain runtime layers, and R3 (executing off-chain CEP applications which, connected to different data sources and sinks, automatically invoke smart contracts when event pattern conditions are met) is fulfilled by off-chain runtime and on-chain runtime layers.

3.1. Off-chain design time layer

The purpose of the off-chain design time layer is to define high-level models of event-driven smart contracts in a way that is understandable to domain experts. These models will be automatically transformed into code, which can then be deployed in both a CEP engine and a blockchain network.

The off-chain design time layer provides the following functionalities. Firstly, the *Smart contract interface/implementation loading* supports loading smart contract code. Interface definitions can be automatically generated in an Application Binary Interface (ABI) definition by existing tools such as web3j (Web3 Labs Ltd, 2020), a Java library for working with smart contracts and integrating with nodes on the Ethereum network, as well as Caterpillar (López-Pintado et al., 2019), a business process execution engine on the Ethereum blockchain that can automatically transform a Business Process Model and Notation (BPMN) process model with Solidity extension into an ABI definition. Alternatively, the implementation of a smart contract can be directly loaded but should be implemented by smart contract programmers. Secondly, *Smart contract graphical modeling* facilitates the graphical design and syntactical validation of smart contract models. More specifically, a smart contract is modeled by specifying its functions with input and output parameters. This graphical modeling can be done manually by a domain expert or automatically by loading a smart contract interface/implementation. Thirdly, the *CEP domain graphical modeling* supports the graphical design and syntactical validation of a CEP domain model, which conforms to the CEP domain metamodel (Boubeta-Puig et al., 2015b). A CEP domain is modeled by specifying the simple event types required for a particular application domain. This graphical modeling can be addressed by a domain expert. Fourthly, the *Event-driven smart contracts graphical modeling* facilitates the design and syntactic validation of event-driven smart contracts in a user-friendly way. An event-driven smart contract is a model defining the event pattern conditions to be satisfied to detect a certain type of situations of interest (complex events). This type of complex event can then be graphically linked to the particular smart contract functions to be automatically invoked upon pattern detection. Finally, there are several types of generation. The *Simple event types automatic generation* automatically transforms the modeled simple event types of a CEP domain into code. The *Event pattern code automatic generation* automatically transforms the modeled pattern conditions into code. The *Smart contract code automatic generation* automatically transforms the modeled event-driven smart contracts into code.

To support all these functionalities, we provide a graphical Domain-Specific Language (DSL) and editor for smart contracts as well as a graphical DSL and editor for event-driven smart contracts (event patterns modeled with smart contracts), as explained below. DSLs are characterized by the following advantages (Fowler & Parsons, 2010): improvements in both the development productivity and communication with domain experts, ease of adaptation to changes and a more rigorous definition, conducted by such experts, of what the system should do.

3.1.1. Graphical DSL and editor for smart contracts

This graphical DSL allows for the definition of smart contracts as graphical models and their syntactic validation. This DSL is composed of two parts. Firstly, the *abstract syntax* consists of both a metamodel, which is a model describing smart contract concepts and relationships between them, and the validation rules to check whether a model is well formed. Secondly, the *concrete syntax* provides a set of useful graphical symbols for drawing smart contract diagrams.

The proposed metamodel for user-friendly definition of smart contracts is illustrated in Fig. 3. This metamodel is composed of various metaclasses. *SmartContracts* is the root metaclass of the metamodel and is composed of a set of one or more smart contracts (*SmartContract*) for an application domain. It is necessary to specify its name (*name*), a textual description (*description*) and creation date (*creationDate*). *SmartContract* describes a particular smart contract. This is the key class since it contains the most sensitive information and is the only one that can be linked by complex events (*ComplexEvent*). Every smart contract has a type name (*typeName*), a path of the image (*imagePath*) that represents it graphically, the private key (*privateKey*) of the blockchain account to be used for deploying or invoking the smart contract, and the contract address (*contractAddress*), which will not be empty if the smart contract was previously deployed in the blockchain network. Moreover, every smart contract will be composed of one or more contract functions (*ContractFunction*). *ContractFunction* describes a function of a smart contract. Every function must have a name (*name*) and, optionally, a path of the image (*imagePath*) that represents it graphically. Additionally, a function can contain one or more input parameters (*InputParameter*), and may have an output parameter (*OutputParameter*). *Parameter* is an

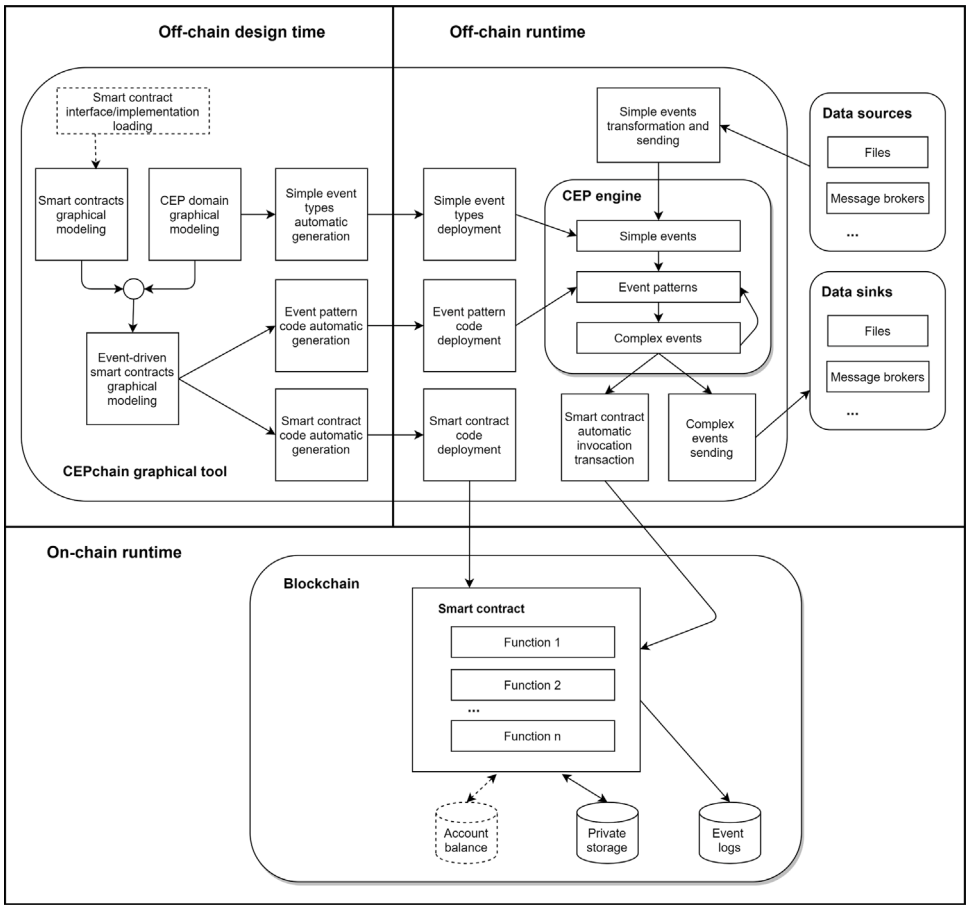

Fig. 2. Overview of the proposed model-driven approach.

abstract metaclass that defines the attributes common to the different types of parameters: parameter name (*name*) and parameter type (*type*). The type will be one of those defined in *PropertyTypeValue*: *Unknown*, *Boolean*, *Integer*, *Long*, *Double*, *Float* or *String*. *InputParameter* represents an input parameter of a smart contract function. It has a value string (*value*). In turn, *OutputParameter* describes the output parameter, i.e. what the function returns.

The validation rules are a fundamental part of the DSL. The following rules must be satisfied by any model that conforms to the metamodel for smart contracts:

- The values of these attributes cannot be empty: *name* and *typeName* of the *SmartContracts* metaclass, *name* of the *ContractFunction* metaclass and *name* of the *Parameter* metaclass.
- The values of these attributes must exist: *typeName* of the *SmartContract* metaclass, and *name* of the *ContractFunction*, *InputParameter* and *OutputParameter* metaclasses.
- The values of the following attributes must have a defined type and must be correct: *type* of the *InputParameter* metaclass and *type* of the *OutputParameter* metaclass.
- The *SmartContract* metaclass is the only one that may have an inbound link connected with a complex event.

This DSL, together with its concrete syntax, was implemented using Eclipse Modeling Framework (EMF) (Steinberg et al., 2008) and its Ecore metamodeling language, as well as the following Eclipse Epsilon languages (Kolovos et al., 2018): Epsilon Object Language (EOL) for creating, querying and modifying EMF models, and Epsilon Validation Language (EVL) for validating models. Additionally, EuGENia, a front-end for Graphical Modeling Framework (GMF), was used to create the graphical modeling editor for this DSL. Fig. 4 depicts the built editor for smart contract definition. This editor has four parts: (i) a tool palette (right panel) from which domain experts can select the elements to be incorporated into their models, (ii) a canvas (central panel) into which users can drag and drop smart contract types along with contract functions from the palette, (iii) a menu (top menu bar) that allows them to easily select the editor action to be executed, and (iv) a property view (lower panel) for adding or editing information related to the different elements of a designed model, for example, the name of a modeled smart contract.

This editor checks that the user uses the palette correctly. As an example, if the user tries to drag and drop an input parameter directly into the canvas, the editor will not allow this to be done since parameters must always be added into contract functions. Moreover, this editor provides the *Smart Contract* menu with the following options. Via *New*, a domain experts can create a new smart contract graphical model. *Load and Model from Caterpillar* automatically creates a new smart contract graphical model by loading an ABI smart contract interface, which is on-demand generated by the Caterpillar tool through the invocation of its REST API (Representational State Transfer Application Programming Interface). *Load and Model from Solidity File* automatically creates a new smart contract graphical model by loading a smart contract implemented in Solidity. Domain experts can also *Deploy Smart Contract* on a blockchain network, *Open* a smart contract model that has already created before, *Save and Validate* the designed model, and *Delete* the current open smart contract model.

3.1.2. Graphical DSL and editor for event-driven smart contracts

To enable the modeling of event patterns together with smart contracts, we extended the ModeL4CEP DSL for event patterns (Boubeta-Puig et al., 2015b) to include our metamodel for smart contracts

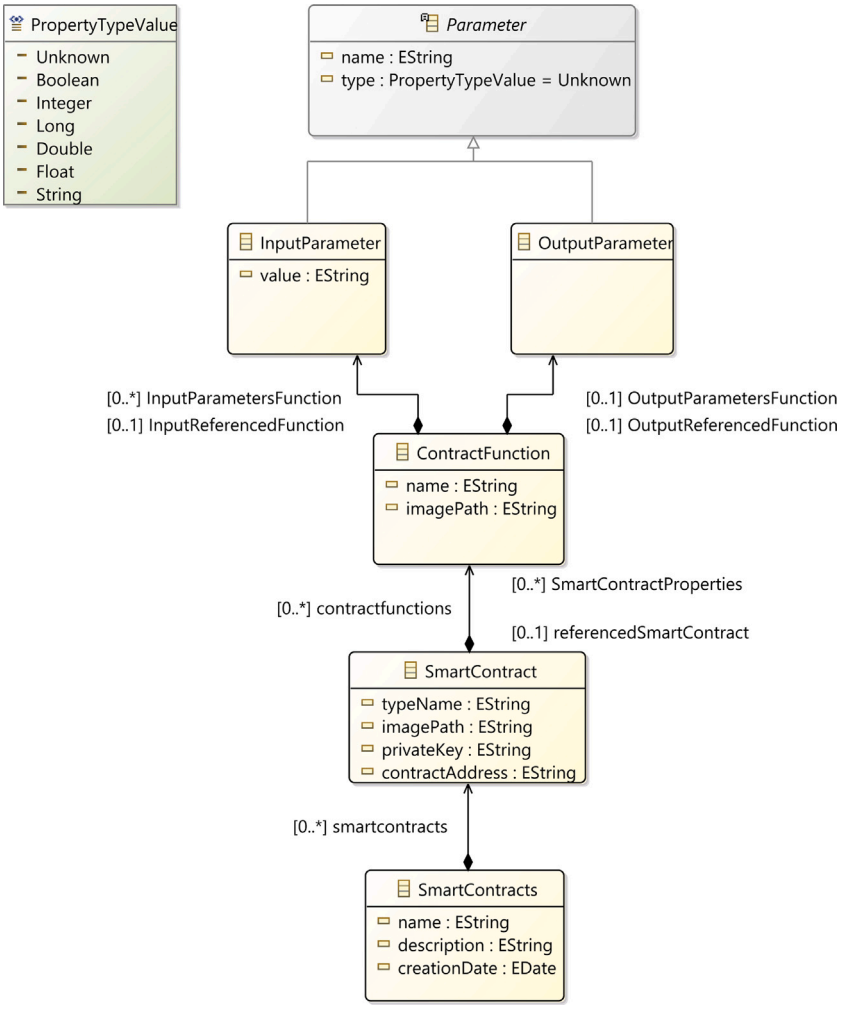

Fig. 3. Metamodel for smart contracts.

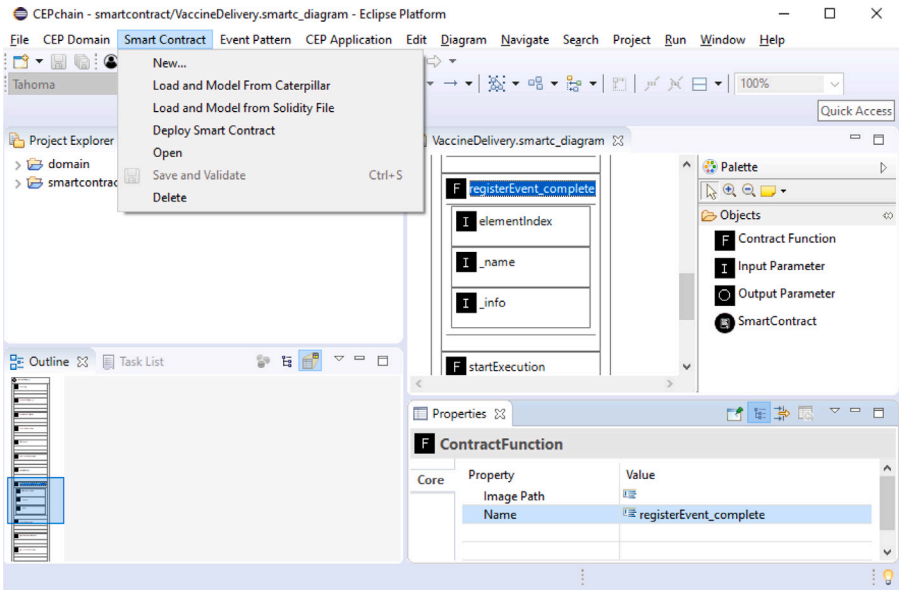

Fig. 4. A screenshot of the graphical modeling editor for smart contracts.

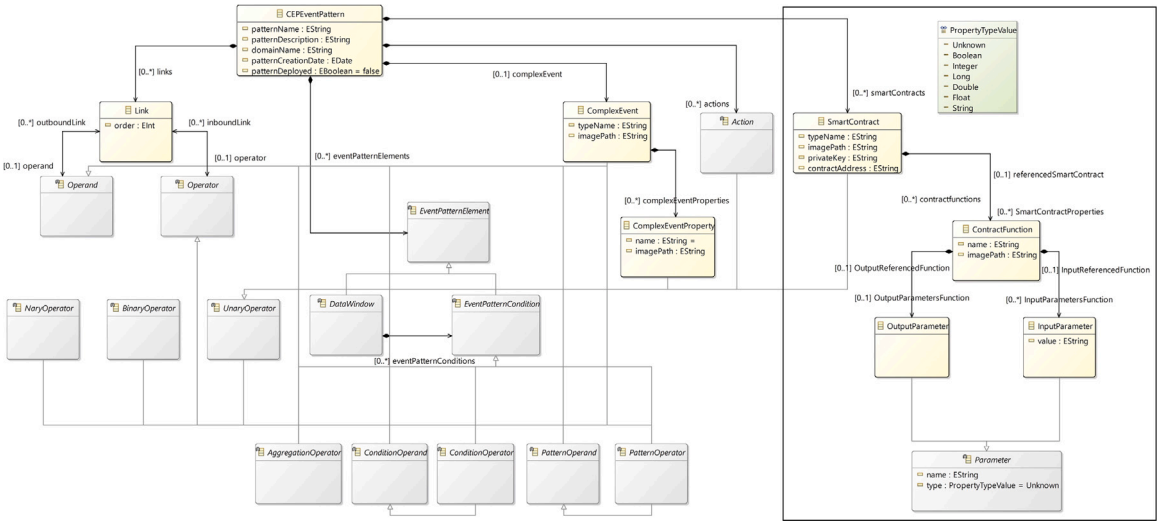

Fig. 5. Metamodel for event-driven smart contracts.

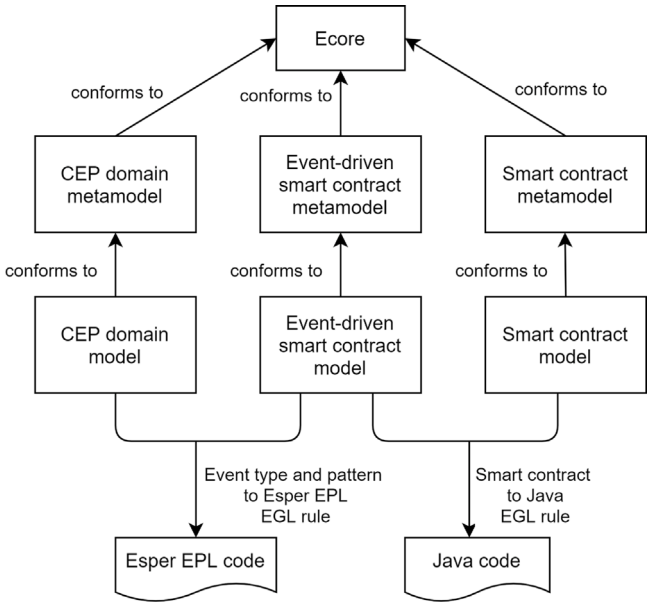

Fig. 6. Model-to-text transformation rules.

presented in Section 3.1.1. More specifically, the *SmartContract* metaclass is included in the ModeL4CEP metamodel as an extra component of the *CEPEventPattern* metaclass (see Fig. 5). This is very powerful since an event pattern can be modeled including a complex event linked to smart contract functions, i.e. upon detection of a complex event, certain smart contract functions will be automatically invoked.

Moreover, using Eclipse Epsilon Generation Language (EGL) and web3j we implemented a set of model-to-text transformation rules for automatically transforming event-driven smart contract models into: (i) the pattern condition implementation in Esper EPL code, which is executable in an Esper CEP engine, (ii) the smart contract implementation in Java code, and (iii) the Java code required to interact with the smart contract in an Ethereum blockchain (see Fig. 6). Note that, since we are using model-driven techniques, other new model-to-text transformation rules could be created to automatically generate code for other CEP engines and blockchain platforms.

The graphical modeling editor for event-driven smart contracts, which is based on MEdit4CEP (Boubeta-Puig et al., 2015a), was also implemented using EMF, the Eclipse Epsilon languages and EuGENia. The key feature of this editor is that its palette can be dynamically reconfigured with both the simple event types and smart contract types previously modeled by the domain expert (see *Simple Events* and *Smart Contracts* tool categories in Fig. 7).

This editor provides the *Event Pattern* menu with the following options. *New* creates a new event pattern model. *Duplicate Existing Pattern* facilitates the creation of a new event pattern model from another previously modeled one. Furthermore, the user can *Open* one of the previously modeled event patterns, *Save and Validate*, and *Delete* the active event pattern model. *Generate Pattern Code* transforms the conditions of the active validated event pattern model into code executable in a CEP engine, while *Generate Smart Contract Code* generates the smart contract code to be automatically invoked by the active validated event pattern model when the conditions are satisfied.

3.2. Off-chain runtime layer

The off-chain runtime layer is responsible for deploying all the code automatically generated from the off-chain design time layer. Specifically, the simple event types and event pattern code are deployed in a CEP engine, while the smart contract code is deployed in a blockchain network at runtime. Fig. 2 shows its four components: (i) data sources, which provide raw and real-time data coming from sensors and other devices and external systems through files, message brokers and other adapters, (ii) a CEP engine, which is the software for analyzing and correlating such data received and transformed into simple events to detect event patterns in real time, (iii) data sinks, which subscribe to and register complex events, generated by the CEP engine, through files, message brokers and other devices and systems, and (iv) the CEPchain graphical tool, which integrates the CEP engine component and functionalities for deploying event type, event pattern and smart contract code. The **Simple event transformation and sending** is in charge of receiving heterogeneous and raw data from data sources, such as Comma-Separated Values (CSV) or JavaScript Object Notation (JSON) files and Message Queue Telemetry Transport (MQTT) or Advanced Message Queuing Protocol (AMQP) message brokers, transforming this data to simple events and sending them to a CEP engine. The **Complex event sending** sends the complex events automatically generated by the CEP engine to data sinks such as CSV or JSON files and MQTT or AMQP message brokers to store and share this meaningful information with other external systems. The **Smart contract automatic invocation transaction** is the key functionality of our approach, which makes it possible to integrate CEP and blockchain. When an event-driven

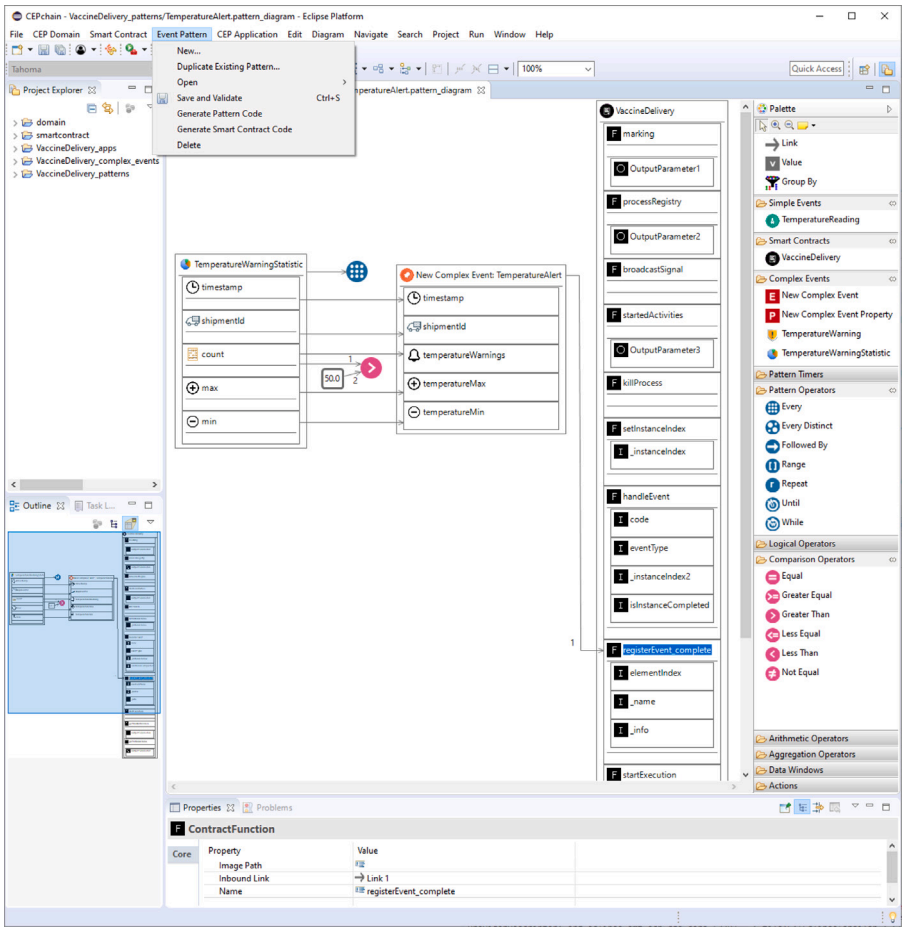

Fig. 7. A screenshot of the graphical modeling editor for event-driven smart contracts.

smart contract previously modeled in the off-chain design time layer is detected by the CEP engine, the corresponding generated complex event will then automatically invoke the specified functions of a smart contract already deployed in a blockchain. So the heavy computation of processing real-time events, reasoning about time and matching event patterns, is performed outside smart contracts, reducing the size and the complexity of smart contracts as well as the cost of their associated transactions (see Section 2).

In order to move towards smart factory applications, it is essential to automate the Business Process Management (BPM) (vom Brocke & Mendling, 2018; Dumas et al., 2018), such as the logistics required to deliver the manufactured products and the payments to be made. This automation can be achieved through the use of smart contracts provided by blockchain technology (Mendling et al., 2018; Preukschat, 2017). Since our model-driven approach allows for integration with the Caterpillar tool, business experts are provided with the CEPchain graphical tool for modeling event patterns connected to the smart contracts that were automatically generated from business processes modeled with Solidity extensions. Therefore, our approach makes it possible to combine BPM, CEP and blockchain.

3.3. On-chain runtime layer

The on-chain runtime layer provides the blockchain in which a smart contract can be deployed through a contract creation transaction. As previously mentioned, in our approach, this type of transaction is currently performed in an Ethereum blockchain by the CEPchain tool.

As illustrated in Fig. 2, a smart contract has functions, a private storage to register its internal state and, optionally, the account balance for cryptocurrency blockchain platforms, such as Ethereum. In our approach, smart contract functions can be executed automatically by complex events that are produced in real time by the Esper CEP engine. Additionally, when an invocation transaction is done, a smart contract can emit events and store event logs in the blockchain. Therefore, this layer supports the two ways data can be stored in Ethereum smart contracts: (i) as a variable in a smart contract, and (ii) as a log event. As previously described in Section 2.1, storing data as log events is cheaper than as variables. However, log events only allow up to three parameters to be queried. In contrast, storing data as a variable is more efficient to manage but is less flexible because of the Solidity constraints on the value types and length.

It should be noted that our approach is appropriate for managing and storing critical situations of interest, i.e. situations that are unlikely to occur. In order to avoid a high cost when automatically invoking transactions and storing data in an Ethereum blockchain, the domain expert, by using the CEPchain graphical tool, can decide and then model which complex event types will invoke smart contract functions and which will not. Thus, the complex event types that are intermediate and not so important for the application domain should not be managed and stored by blockchain. Evidently, the domain expert could model all complex event types linked to smart contracts, but assuming extra gas costs and time consumption.

Since we are using a model-driven approach, event-driven smart contract models are independent of specific blockchain platforms. Therefore, by adding new model-to-text transformation rules, the generated code could be deployed in permissioned blockchain platforms, such as Hyperledger Fabric (Linux Foundation, 2021), in which gas cost is not taken into consideration.

Fig. 8. Vaccine delivery scenario.

4. Evaluation

The purpose of this evaluation is to demonstrate that the CEPchain approach fulfills the requirements established in Section 1. To this end, our implementation is applied to a real-world case study and the results are then obtained and discussed.

4.1. Case study description

Blockchain technology can be used in a variety of industrial sectors, such as transport, healthcare, manufacturing, tourism and agriculture. Supply chains are one of the well-known applications in which the movement of goods between participants is critical. By using smart contracts, the execution of the supply chain process can be controlled. The key events produced as a consequence of tracking such goods can be registered and communicated through data stored on a blockchain. Thus, supply chain quality and logistics visibility can be provided and improved (Xu et al., 2019).

In particular, we adopt CEPchain for a vaccine supply chain scenario. Fig. 8, adapted from Fournier and Skarbovsky (2019), shows all the steps typically involved in a vaccine supply chain. Firstly, pharmaceutical companies deliver vaccines to both pharmacies and hospitals through distributors. Next, these pharmacies and hospitals distribute them to doctors through local distributors. Finally, doctors administer vaccines to patients.

One of the main challenges to be addressed during the transport of vaccines is monitoring temperature conditions in real time to promptly detect when they are exposed to temperatures above or below those recommended by the World Health Organization (2006), as excessive exposure to heat or cold can render vaccines unusable. To address the real-time detection of abnormal vaccine temperatures and the improvement of logistics visibility through the storage of key temperature events on blockchain, we use our model-driven approach integrating CEP and blockchain.

In this scenario, vaccines are transported in containers that have a unique identifier and are equipped with a temperature sensor sending a reading every minute. Different event patterns can be identified. Firstly, *TemperatureWarning* is a pattern that detects when a temperature sensor value is outside the 2–8 °C range, warning that the current temperature of a vaccine is not that recommended by the World Health Organization (2006). Secondly, *TemperatureWarningStatistic* calculates the number of *TemperatureWarning* complex events produced, as well as the maximum and minimum temperature values per hour. Thirdly, *TemperatureAlert* detects when vaccines have been exposed to inadequate temperatures on several occasions over a short period of time. Specifically, this pattern checks whether more than 50 *TemperatureWarning* complex events are detected during an hour, which means the vaccine might be rendered obsolete. If this occurs, the generated *TemperatureAlert* is a key event in our application domain, and this complex event will automatically invoke a smart contract in charge of storing critical complex events in the blockchain.

4.2. Results

Next, considering the case study description, we demonstrate that CEPchain fulfills the requirements established in Section 1. Please note that all files used and obtained in this vaccine delivery case study, conducted to test our CEPchain model-driven solution, have been published as a dataset in the Mendeley Data repository (Boubeta-Puig et al., 2021).

4.2.1. R1: Graphically modeling event-driven smart contracts

In order to graphically model event-driven smart contracts in a graphical way, as previously explained in Section 3 and illustrated in Fig. 2, the following steps were followed. Firstly, our CEPchain graphical tool allowed us to load the interface of a smart contract and then automatically model it in a graphical way by using the *Load and Model from Caterpillar* menu option. In this case study, this interface was automatically generated from the BPMN process model with Solidity extension modeled using the Caterpillar tool (see Fig. 9). So, Caterpillar generated the *VaccineDelivery* smart contract, which captures the underlying behavior of the modeled BPMN process, and the CEPchain tool automatically designed its interface as a model and validated it (see Fig. 4).

This smart contract contains a function, generated and named as *registerEvent_complete* by Caterpillar, which has the following three input parameters. The *_name* parameter is the name of the complex event type to be automatically registered in the blockchain, e.g. *TemperatureAlert*. The *_info* parameter refers to all information about the complex event to be registered, i.e. each property of the event with its value, for example: "*timestamp: 1591983419, shipmentId: id1, temperatureWarnings: 53, temperatureMax: 15.2, temperatureMin: -1.2*". The *elementIndex* parameter is internally used by Caterpillar to refer a particular BPMN element.

We then graphically designed the CEP domain model for our case study, containing the *TemperatureReading* simple event type (see Fig. 10). This event type has three properties: the *timestamp* (in epoch format) in which the temperature was taken by a sensor located in a shipment container; the unique *shipmentId* of a particular shipment, which involves the transport of several vaccines; and the *temperature* value measured at a specific time instant.

Once the smart contract and CEP domain had been modeled, the *TemperatureWarning*, *TemperatureWarningStatistic* and *TemperatureAlert* event patterns were modeled with the CEPchain tool (see Figs. 10, 11 and 7, respectively). As illustrated in Fig. 7, this is an event-driven smart contract in which *TemperatureAlert* complex events, generated by a CEP engine at runtime, can themselves automatically invoke the *registerEvent_complete* function of the *VaccineDelivery* smart contract previously deployed in a blockchain.

Therefore, we can state that our CEPchain tool satisfies requirement R1, i.e. graphically modeling event-driven smart contracts.

4.2.2. R2: Automatically transforming event-driven smart contracts models into code and then deploying this code in both a CEP engine and a blockchain network

Once the interface of the *VaccineDelivery* smart contract (see Fig. 4) had been modeled, it was automatically generated into code and stored in the *ContractName*.java file. This file contains definitions of all the functions that can be invoked. Table 3 shows the total number of Lines of Code (LoC) automatically generated by the CEPchain tool.

To conduct the automatic deployment of a smart contract, a new Java file, named *ContractName*_deploy.java, was automatically generated. This file contains only the deployment function. This file was then automatically invoked, with a contract address being assigned to this smart contract by the Ethereum blockchain (in this case, `0x0375b5851AAfd27c0DD4a52C1E06D5641480Fa5d`). Subsequently, this address was automatically registered in the graphical contract model. Note that this address is needed to later invoke the smart contract.

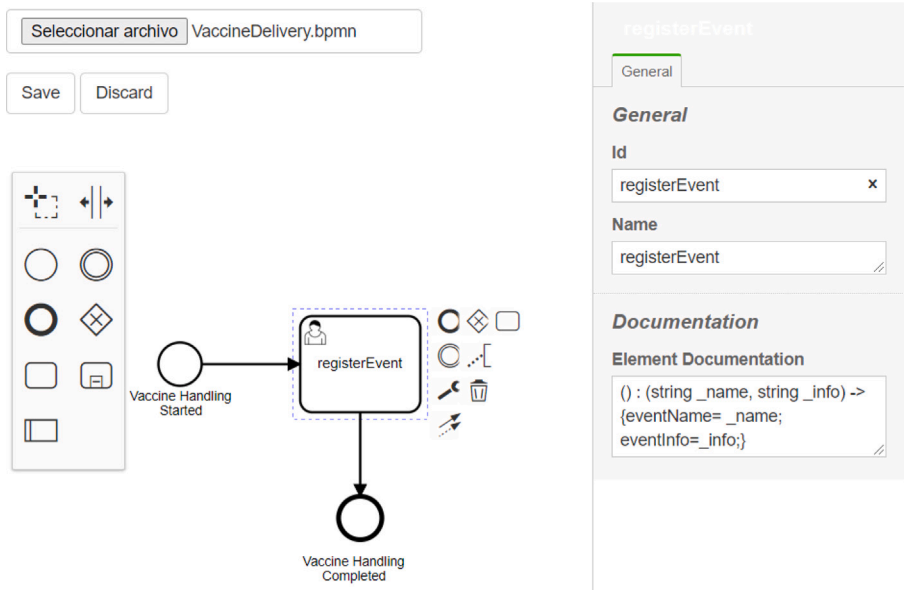

Fig. 9. *VaccineDelivery* BPMN process with Solidity extension modeled with the Caterpillar tool.

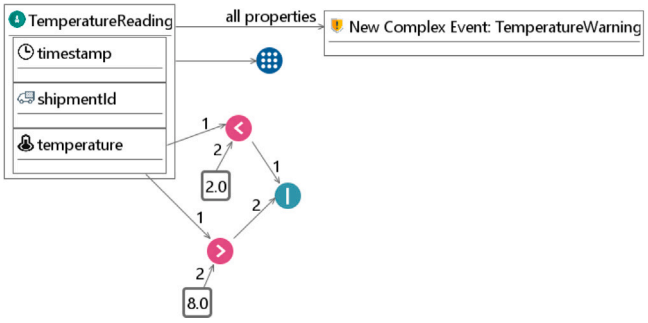

Fig. 10. *TemperatureWarning* event pattern modeled with the CEPchain tool.

Table 3
LoCs per file automatically generated by the CEPchain tool.

File generated	LoC generated
VaccineDelivery.java	208
VaccineDelivery_deploy.java	73
VaccineDelivery_invocation_TemperatureAlert.java	67
TemperatureWarning.epl	5
TemperatureWarningStatistic.epl	10
TemperatureAlert.epl	9

To address the automatic invocation, the event-driven smart contract model was automatically transformed into Java code. The *ContractName*_invocation_ *PatternName*.java contains the implementation of the functions to be invoked, i.e. the functions that were graphically linked by a complex event in the event pattern model.

Additionally, every event pattern model was automatically transformed into Esper EPL code and then deployed in an Esper engine. Table 3 also shows the Esper EPL files and LoCs automatically generated by the CEPchain tool for this case study. Therefore, our graphical modeling tool also fulfills requirement R2, i.e. automatically transforming event-driven smart contracts models into code and then deploying this code in both a CEP engine and a blockchain network.

4.2.3. R3: Executing off-chain CEP applications which, connected to different data sources and sinks, automatically invoke smart contracts when event pattern conditions are met

Once the event patterns are graphically modeled (see Figs. 7, 10 and 11), the CEPchain tool allows domain experts to define off-chain CEP applications. These applications are composed of data sources, event patterns and data sinks. As explained in Section 3, data sources can be files or message queues in charge of receiving external raw and heterogeneous data in formats such as CSV and JSON. These data are then transformed and sent to the Esper CEP engine. By making use of event patterns, the CEP engine can detect situations of interest (complex events), and also invoke the smart contract functions specified in the event-driven smart contract models. These complex events are then notified through data sinks, such as files and message queues, which could be connected to external applications like dashboards or mobile apps.

To demonstrate that our solution allows domain experts to execute CEP applications integrating data sources, event-driven smart contracts and data sinks, firstly, we implemented a Java simulator that generates

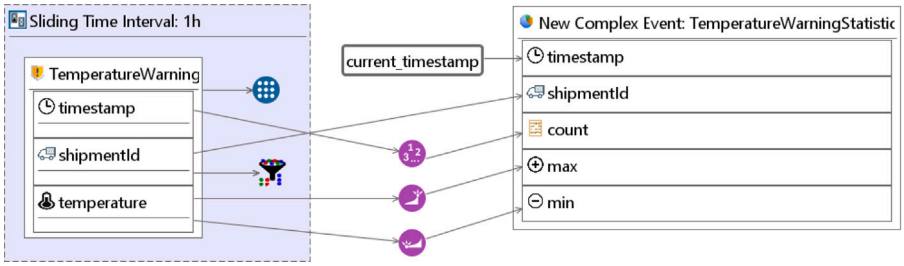

Fig. 11. *TemperatureWarningStatistic* event pattern modeled with the CEPchain tool.

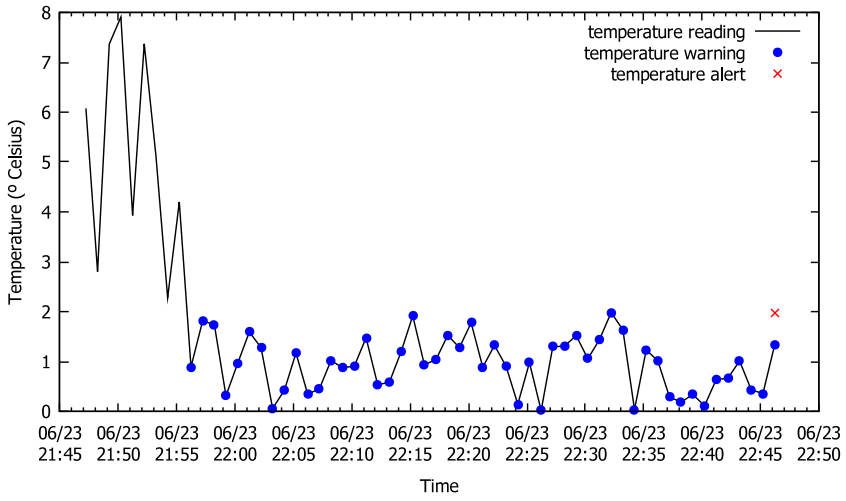

Fig. 12. Simple and complex events detected automatically from shipment id 5 by the Esper engine.

Table 4
Simple and complex events detected automatically by the Esper engine for the vaccine delivery case study.

Event name	Event type	Detected events
TemperatureReading	Simple	300
TemperatureWarning	Complex	102
TemperatureWarningStatistic	Complex	102
TemperatureAlert	Complex	2

temperature readings from five different vaccine shipments (*id1*, *id2*, *id3*, *id4*, *id5*) during an hour. Temperature values are generated randomly, normally between 2 and 8 °C; from time to time, a temperature value is generated out of this range.

We then defined a CEP application that received such temperature readings generated by the simulator and transformed them into *TemperatureReading* simple events. By using such modeled event patterns, the CEP engine generated *TemperatureWarning*, *TemperatureWarningStatistic* and *TemperatureAlert* complex events in real time. Table 4 shows the number of the simple events and complex events automatically generated by the Esper CEP engine. These complex events were then stored in CSV files (once per complex event type). Fig. 12 depicts the simple events (*TemperatureReading*) and complex events (*TemperatureWarning* and *TemperatureAlert*) detected automatically from shipment id 5 by the Esper engine. In particular, the following events were detected: 60 *TemperatureReading* simple events (once every 1 min during 1 h), 51 *TemperatureWarning* complex events for temperature readings lower than 2 °C, and 1 *TemperatureAlert* since more than 50 *TemperatureWarning* events were previously generated.

Upon every *TemperatureAlert* pattern detection, the *registerEvent_complete* function of the smart contract previously deployed was automatically invoked by the Esper engine. Fig. 13 illustrates that this function was in fact invoked twice, storing the two critical detected *TemperatureAlert* complex events in the Ethereum blockchain. Therefore, our graphical modeling tool also fulfills requirement R3, i.e. executing off-chain CEP applications which, connected to different data sources and sinks, automatically invoke smart contracts when event pattern conditions are met.

As a result, we can conclude that CEPchain, our model-driven approach integrating CEP and blockchain, allows end users to address the real-time detection of abnormal vaccine temperatures and the improvement of logistics visibility through the automatic storage of critical temperature events on the public Ethereum blockchain.

4.3. Discussion

Our model-driven approach integrating CEP and blockchain is appropriate for managing and storing immutably critical situations of interest, i.e. situations that are unlikely to occur. As an example, and as previously shown, CEPchain is adequate for vaccine delivery, since this case study requires real-time data to be processed (temperatures) over time in order to detect critical situations of interest. This can be addressed by using the processing power of a CEP engine. We can assume that a temperature alert, meaning that a particular vaccine might be unusable from that moment, will rarely happen. Thus, managing and storing these few alert events in blockchain will be neither time-consuming nor expensive in terms of gas.

Moreover, CEPchain is a no-code platform, providing end users with the ability to graphically define event-driven smart contracts (event patterns in which their complex events are linked to smart contract functions), without requiring the implementation of any code. This tool can then automatically transform the defined graphical models into error-free code. For example, for the vaccine delivery case study, 6 files needed to be automatically generated with a total of 372 LoCs: 24 LoCs were needed to implement the event patterns in Esper EPL, while 348 LoCs were required to implement the *VaccineDelivery* smart contract and its invocation in Java code. It is worth noting that if CEPchain were not used, then end users would need to be experts not only in Esper EPL (or other EPLs), but also in Solidity (or even other smart contract languages). Although we might find an expert in both CEP and blockchain, this expert could then make errors when implementing event-driven smart contracts. Note that a simple error in the implementation of a smart contract deployed in an Ethereum blockchain could imply a significant loss of money.

Implementing the required logic of this case study by using our model-driven approach has various benefits. Firstly, design becomes feasible for domain experts. When defining event patterns, users do not have to explicitly implement the sequence of steps to define what has to be done. However, by using Solidity, programmers have to write what must be done and how to obtain it. Secondly, errors can be avoided. Event patterns can be graphically modeled and the generated code is validated syntactically, facilitating changes in models. In contrast, the order of instructions manually implemented in Solidity can affect the correctness of the obtained smart contract. Thirdly, we obtain expressiveness for time and events. By using CEPchain, the logic of event-driven smart contracts can be more powerful and complete compared to the logic defined by using Solidity, as Solidity currently has the following limitations: (i) long type is not supported, being needed for

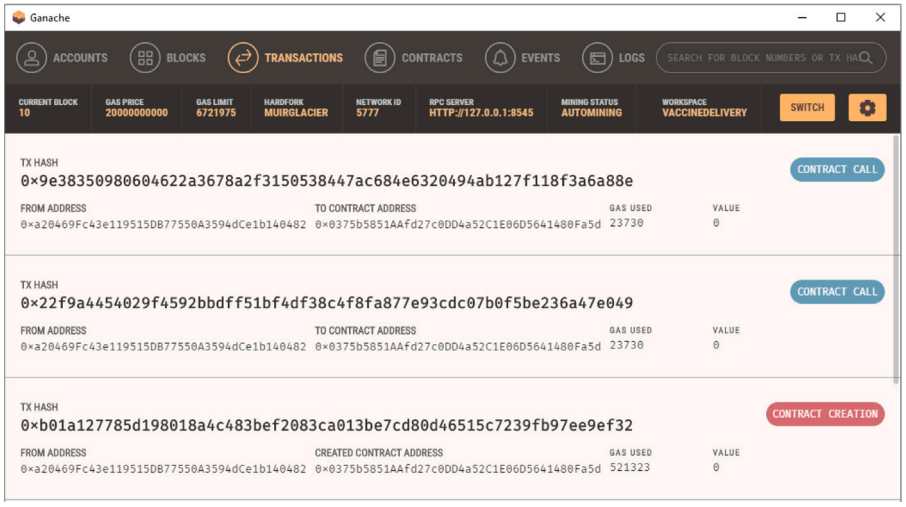

Fig. 13. *VaccineDelivery* smart contract creation transaction, and invocation transactions automatically executed upon event pattern detection.

real-time timestamp; (ii) fixed point numbers are not fully supported; they can be declared but assigned to or from (e.g. a temperature value of 37.5 should be stored as 375); (iii) aggregation functions as *count()*, *max()* and *min()* are not supported, thus complicating the definition of logic for detecting temperature warning statistics; and (iv) data windows and temporal operators are not supported, so it is not possible to detect situations of interest dealing with temporal restrictions/aspects.

According to the study conducted for MEdit4CEP (Boubeta-Puig et al., 2015a), on which CEPchain is based, this tool has an appropriate level of usability for the following two groups of users: experts in an application domain but not in CEP, and experts in both a domain and CEP. Indeed, most users reported that the graphical tool could notably reduce the time needed to define event patterns, instead of implementing them manually by using a specific EPL. As CEPchain allows domain experts to load and automatically model a smart contract thanks to its integration with Caterpillar, these users can model event-driven smart contracts by means of drag and drop. This means that their validation, model-to-text transformation and deployment in both a CEP engine and a blockchain are transparently performed.

5. Conclusions and future work

In this paper, we proposed the integration of the CEP and blockchain technologies through a model-driven solution, CEPchain, for graphical event-driven smart contract design, automatic code generation and execution in both a CEP engine and a blockchain network. CEPchain allows domain experts to load and automatically model a smart contract thanks to its integration with the Caterpillar tool, giving support for modeling business process declarations as graphical models and transforming them into smart contract models. These users can then graphically design event patterns in which such smart contracts can be associated as actions to be carried out upon pattern condition detection. Pattern conditions are automatically transformed into Esper EPL code, which is deployed and executed on an Esper CEP engine, and smart contracts are transformed into Java code, which is deployed and executed on an Ethereum blockchain. Therefore, this approach supports event-driven smart contracts, i.e. smart contract functions can be automatically invoked by complex events generated in real time.

This model-driven solution was validated through a real-world case study for vaccine delivery ensuring the cold chain. More specifically, this case study highlights that CEPchain is able to process and correlate temperature readings taken from shipment containers with the aim of detecting temperature alerts in real time, indicating that certain vaccines would become obsolete as a result of the loss of the cold chain. This information is then registered in a blockchain, making this vaccine supply transparent and traceable for any user. The results showed that this proposal is adequate for integrating CEP and blockchain through a graphical model-driven tool and can be applied to different application domains without requiring experts in event processing and smart contract languages.

As future work, we plan to create new model-to-text rules for transforming the modeled event-driven smart contracts into other programming languages such as Visual Studio and JavaScript, allowing their deployment on other blockchain platforms like IBM Blockchain Platform (Gupta, 2018) and Hyperledger Fabric.

CRediT authorship contribution statement

Juan Boubeta-Puig: Conceptualization, Methodology, Software, Validation, Investigation, Resources, Writing – original draft, Visualization, Funding acquisition. **Jesús Rosa-Bilbao:** Methodology, Software, Validation, Investigation, Resources, Writing – original draft. **Jan Mendling:** Conceptualization, Methodology, Investigation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the Spanish Ministry of Science and Innovation under the “Estancias de movilidad en el extranjero José Castillejo para jóvenes doctores” program [grant number CAS19/00241], and the Spanish Ministry of Science and Innovation and the European Regional Development Funds under project FAME [grant number RTI2018-093608-B-C33]. The authors would like to thank Orlenys López-Pintado for his help with the Caterpillar tool and his insightful comments. Juan Boubeta-Puig would also like to thank the Institute for Information Business for their hospitality when visiting

them at the Vienna University of Economics and Business, Austria, where part of this work was developed.

References

Ali, M. S., Vecchio, M., Pincheira, M., Dolui, K., Antonelli, F., & Rehmani, M. H. (2019). Applications of blockchains in the internet of things: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, *21*(2), 1676–1717. http://dx.doi.org/10.1109/COMST.2018.2886932.

Boubeta-Puig, J., Ortiz, G., & Medina-Bulo, I. (2014). A model-driven approach for facilitating user-friendly design of complex event patterns. *Expert Systems with Applications*, *41*(2), 445–456. http://dx.doi.org/10.1016/j.eswa.2013.07.070.

Boubeta-Puig, J., Ortiz, G., & Medina-Bulo, I. (2015). MEdit4CEP: A model-driven solution for real-time decision making in SOA 2.0. *Knowledge-Based Systems*, *89*, 97–112. http://dx.doi.org/10.1016/j.knosys.2015.06.021.

Boubeta-Puig, J., Ortiz, G., & Medina-Bulo, I. (2015). ModeL4CEP: Graphical domain-specific modeling languages for CEP domains and event patterns. *Expert Systems with Applications*, *42*(21), 8095–8110. http://dx.doi.org/10.1016/j.eswa.2015.06.045.

Boubeta-Puig, J., Rosa-Bilbao, J., & Mendling, J. (2021). *Dataset for CEPchain: A graphical model-driven solution for integrating complex event processing and blockchain*. Mendeley Data, v1, http://dx.doi.org/10.17632/s8fhhrfrzg.1.

Brambilla, M., Cabot, J., & Wimmer, M. (2017). *Model-driven software engineering in practice* (second ed.). Morgan & Claypool Publishers.

vom Brocke, J., & Mendling, J. (Eds.), (2018). *Management for professionals*, *Business process management cases*. Cham: Springer International Publishing, http://dx.doi.org/10.1007/978-3-319-58307-5.

Burgueño, L., Boubeta-Puig, J., & Vallecillo, A. (2018). Formalizing complex event processing systems in Maude. *IEEE Access*, *6*, 23222–23241. http://dx.doi.org/10.1109/ACCESS.2018.2831185.

Calvo, I., Merayo, M. G., & Núñez, M. (2019). A methodology to analyze heart data using fuzzy automata. *Journal of Intelligent & Fuzzy Systems*, *37*(6), 7389–7399. http://dx.doi.org/10.3233/JIFS-179348.

Chandy, K., & Schulte, W. (2010). *Event processing: designing IT systems for agile companies* (first ed.). New York, USA: McGraw-Hill, Inc..

Cummins, F. A. (2009). Chapter 8 - Event-driven agility. In F. A. Cummins (Ed.), *The MK/OMG Press*, *Building the agile enterprise* (pp. 207–229). Burlington: Morgan Kaufmann, http://dx.doi.org/10.1016/B978-0-12-374445-6.00008-X.

Díaz, G., Macià, H., Valero, V., Boubeta-Puig, J., & Cuartero, F. (2020). An Intelligent Transportation System to control air pollution and road traffic in cities integrating CEP and Colored Petri Nets. *Neural Computing and Applications*, *32*(2), 405–426. http://dx.doi.org/10.1007/s00521-018-3850-1.

Dumas, M., La Rosa, M., Mendling, J., & Reijers, H. A. (2018). *Fundamentals of business process management*. Berlin, Heidelberg: Springer Berlin Heidelberg, http://dx.doi.org/10.1007/978-3-662-56509-4.

Eshuis, R., Norta, A., & Roulaux, R. (2016). Evolving process views. *Information and Software Technology*, *80*, 20–35. http://dx.doi.org/10.1016/j.infsof.2016.08.004.

EsperTech (2021). Esper. http://www.espertech.com/esper/, (Accessed 21 April 2021).

Ethereum (2021). Solidity 0.6.8 documentation. https://solidity.readthedocs.io/en/v0.6.8/, (Accessed 21 April 2021).

Ethereum Foundation (2021). Ethereum. https://ethereum.org, (Accessed 21 April 2021). Ethereum.Org.

Etzion, O., & Niblett, P. (2010). *Event processing in action* (first ed.). Greenwich, CT, USA: Manning Publications Co..

Event Processing Technical Society (2011). Event processing glossary - version 2.0. http://www.complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Processing_Glossary_v2.pdf, (Accessed 21 April 2021).

Farrugia, S., Ellul, J., & Azzopardi, G. (2020). Detection of illicit accounts over the Ethereum blockchain. *Expert Systems with Applications*, *150*, Article 113318. http://dx.doi.org/10.1016/j.eswa.2020.113318.

Fournier, F., & Skarbovsky, I. (2019). Enriching smart contracts with temporal aspects. In J. Joshi, S. Nepal, Q. Zhang, & L.-J. Zhang (Eds.), *Lecture notes in computer science*, *Blockchain* (pp. 126–141). Cham: Springer International Publishing, http://dx.doi.org/10.1007/978-3-030-23404-1_9.

Fowler, M., & Parsons, R. (2010). *Addison-Wesley signature*, *Domain specific languages* (first ed.). Massachussets, USA: Addison Wesley.

Gartner (2020). Blockchain technology & how it helps business growth. https://www.gartner.com/en/information-technology/insights/blockchain, (Accessed 21 April 2021).

Gupta, M. (2018). *Blockchain for dummies* (second ed.). New Jersey, USA: John Wiley & Sons, Inc..

Hinze, A., Sachs, K., & Buchmann, A. (2009). Event-based applications and enabling technologies. In *Proceedings of the third ACM international conference on distributed event-based systems* (pp. 1–15). Nashville, Tennessee: Association for Computing Machinery, http://dx.doi.org/10.1145/1619258.1619260.

Idelberger, F., Governatori, G., Riveret, R., & Sartor, G. (2016). Evaluation of logic-based smart contracts for blockchain systems. In J. J. Alferes, L. Bertossi, G. Governatori, P. Fodor, & D. Roman (Eds.), *Lecture notes in computer science*, *Rule technologies. Research, tools, and applications* (pp. 167–183). Cham: Springer International Publishing, http://dx.doi.org/10.1007/978-3-319-42019-6_11.

Kolovos, D., Rose, L., García-Domínguez, A., & Paige, R. (2018). *The Epsilon book*. URL: http://www.eclipse.org/epsilon/doc/book/. (Accessed 21 April 2021).

Kshetri, N. (2018). Blockchain and electronic healthcare records [cybertrust]. *Computer*, *51*(12), 59–63. http://dx.doi.org/10.1109/MC.2018.2880021.

Kshetri, N., & Voas, J. (2018). Blockchain-enabled E-voting. *IEEE Software*, *35*(4), 95–99. http://dx.doi.org/10.1109/MS.2018.2801546.

Linux Foundation (2021). Hyperledger fabric. https://www.hyperledger.org/use/fabric, (Accessed 21 April 2021).

López-Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I., & Ponomarev, A. (2019). Caterpillar: A business process execution engine on the Ethereum blockchain. *Software - Practice and Experience*, *49*(7), 1162–1193. http://dx.doi.org/10.1002/spe.2702.

Luckham, D. (2012). *Event processing for business: organizing the real-time enterprise*. New Jersey, USA: John Wiley & Sons.

Mendling, J., Weber, I., Aalst, W. V. D., Brocke, J. V., Cabanillas, C., Daniel, F., Debois, S., Ciccio, C. D., Dumas, M., Dustdar, S., Gal, A., García-Bañuelos, L., Governatori, G., Hull, R., Rosa, M. L., Leopold, H., Leymann, F., Recker, J., Reichert, M., ... Zhu, L. (2018). Blockchains for business process management - Challenges and opportunities. *ACM Transactions on Management Information Systems (TMIS)*, *9*(1), 4–16. http://dx.doi.org/10.1145/3183367.

Moin, S., Karim, A., Safdar, Z., Safdar, K., Ahmed, E., & Imran, M. (2019). Securing IoTs in distributed blockchain: Analysis, requirements and open issues. *Future Generation Computer Systems*, *100*, 325–343. http://dx.doi.org/10.1016/j.future.2019.05.023.

Mühlberger, R., Bachhofner, S., Di Ciccio, C., García-Bañuelos, L., & López-Pintado, O. (2019). Extracting event logs for process mining from data stored on the blockchain. In C. Di Francescomarino, R. Dijkman, & U. Zdun (Eds.), *Lecture notes in business information processing*, *Business process management workshops* (pp. 690–703). Cham: Springer International Publishing, http://dx.doi.org/10.1007/978-3-030-37453-2_55.

Preukschat, A. (2017). *Blockchain: la revolución industrial de internet*. Barcelona: Gestión 2000.

Roldán, J., Boubeta-Puig, J., Martínez, J. L., & Ortiz, G. (2020). Integrating complex event processing and machine learning: An intelligent architecture for detecting IoT security attacks. *Expert Systems with Applications*, Article 113251. http://dx.doi.org/10.1016/j.eswa.2020.113251.

Skarbovsky, I. (2020). Proton. https://github.com/ishkin/Proton, (Accessed 21 April 2021).

Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2008). *Eclipse series*, *EMF: Eclipse modeling framework* (second ed.). Addison-Wesley Professional.

Web3 Labs Ltd (2020). Web3j. https://docs.web3j.io/, (Accessed 21 April 2021).

Wood, G. (2019). Ethereum: a secure decentralized generalised transaction ledger. *GitHub*, 1–39, URL: https://github.com/ethereum/yellowpaper.

World Health Organization (2006). *Temperature sensitivity of vaccines*: *Technical report*, (WHO/IVB/06.10), (p. 73). Geneva, Switzerland: World Health Organization, URL: https://apps.who.int/iris/handle/10665/69387.

WSO2 (2020). Siddhi. https://siddhi.io/, (Accessed 21 April 2021).

Xu, X., Weber, I., & Staples, M. (2019). *Architecture for blockchain applications*. Cham, Switzerland: Springer International Publishing, http://dx.doi.org/10.1007/978-3-030-03035-3.

Yaga, D. J., Mell, P. M., Roby, N., & Scarfone, K. (2018). *Blockchain technology overview*: *NIST pubs*, (8202), (pp. 1–66). Gaithersburg, MD: NIST, http://dx.doi.org/10.6028/NIST.IR.8202.

Yang, X., Yi, X., Nepal, S., Kelarev, A., & Han, F. (2020). Blockchain voting: Publicly verifiable online voting protocol without trusted tallying authorities. *Future Generation Computer Systems*, *112*, 859–874. http://dx.doi.org/10.1016/j.future.2020.06.051.

Juan Boubeta-Puig is a tenured Associate Professor with the Department of Computer Science and Engineering at the University of Cadiz (UCA), Spain. He received his Ph.D. in Computer Science from UCA in 2014 and was honored with the Extraordinary Ph.D. Award from UCA and the Best Ph.D. Thesis Award from the Spanish Society of Software Engineering and Software Development Technologies. His research interests include real-time big data analytics through complex event processing, event-driven service-oriented architecture, Internet of things, blockchain and model-driven development of advanced user interfaces, and their application to e-health, smart city, industry 4.0 and cybersecurity.

Jesús Rosa-Bilbao is a Ph.D. student in the UCASE Software Engineering Research Group at the University of Cadiz (UCA), Spain. He received his B.Sc. degree in Computer Science specialized in Software Engineering from UCA in 2019. In addition, he received his M.Sc. degree in Cybersecurity from UCA, his M.Sc. degree in Project Management from the European Business School in Barcelona (ENEB) and his M.Sc. degree in Business Administration from ENEB in 2020. His research interests include complex event processing, event-driven service-oriented architecture, business process modeling, blockchain and cybersecurity.

Jan Mendling is a Full Professor at Wirtschaftsuniversität Wien, Austria. His research interests include business process management and information systems. He has published more than 400 research papers and articles, among others in ACM Transactions on Software Engineering and Methodology, IEEE Transaction on Software Engineering, Information Systems, European Journal of Information Systems, and Decision Support Systems. He is a board member of the Austrian Society for Process Management, one of the founders of the Berliner BPM-Offensive, and member of the IEEE Task Force on Process Mining. He is a co-author of the textbooks Fundamentals of Business Process Management and Wirtschaftsinformatik.