DMN Decision Execution on the Ethereum Blockchain

Stephan Haarmann(✉), Kimon Batoulis, Adriatik Nikaj, and Mathias Weske

Hasso Plattner Institute, University of Potsdam, Potsdam, Germany
stephan.haarmann@student.hpi.de,
{kimon.batoulis,adriatik.nikaj,mathias.weske}@hpi.de

Abstract. Recently blockchain technology has been introduced to execute interacting business processes in a secure and transparent way. While the foundations for process enactment on blockchain have been researched, the execution of decisions on blockchain has not been addressed yet. In this paper we argue that decisions are an essential aspect of interacting business processes, and, therefore, also need to be executed on blockchain. The immutable representation of decision logic can be used by the interacting processes, so that decision taking will be more secure, more transparent, and better auditable. The approach is based on a mapping of the DMN language S-FEEL to Solidity code to be run on the Ethereum blockchain. The work is evaluated by a proof-of-concept prototype and an empirical cost evaluation.

Keywords: Blockchain · Interacting processes · DMN

1 Introduction

Business processes are an acknowledged means to describe working procedures in organizations [1]. Activities, their ordering, the data they work on, and organizational responsibilities can be represented in business processes, for instance using the Business Process Model and Notation (BPMN) [2]. Recently, BPMN has been accompanied by the Decision Model and Notation (DMN) [3] standard to capture decision structure and decision logic, thereby achieving a separation of concerns of process and decision logic [4]. More and more companies use both standards to describe business processes and decisions taken by them. Process enactment architectures are enhanced by decision engines that are capable of executing decision logic during process executions.

While we currently witness a strong uptake of decision management in industry, decision models are almost exclusively used to represent internal decisions of a company, such as whether a specific credit can be granted or not. However, many decisions are linked to more than one process and are part of interacting business processes. Such processes have been subject to behavioral analysis [1,5].

To address the security and transparency needs of interacting business processes, recently blockchain technology was proposed as an enactment platform for interacting business processes [6]. Blockchains enable the execution of interacting processes through software code to be run on a blockchain – so-called smart contracts – even if the participants do not trust each other.

While this work introduces blockchain technology to execute interacting processes, it does not address decisions that are taken by them. To close this gap, in this paper we propose an approach to execute DMN decision models on an Ethereum blockchain, thereby improving security, transparency, and auditing of decisions taken by multiple interacting business processes. The approach is based on a mapping of DMN decision models expressed in the language S-FEEL to Solidity, the programming language of the Ethereum blockchain [7].

The remainder of this paper is structured as follows. In Sect. 2 we provide an overview of DMN and of relevant aspects of blockchain technology. Section 3 contains the mapping of DMN decision models to Ethereum smart contracts. The approach is evaluated with respect to the blockchain specific costs in Sect. 4. An overview of related work is provided in Sect. 5. We discuss our research in Sect. 6.

2 Background

DMN decision models express requirements and logic of decisions. While the execution of DMN in process orchestrations can be achieved by traditional software systems, such as rule engines, decisions taken during interacting processes have not been investigated. We claim that blockchain technology proves to be useful in such settings due to a set of properties, which are provided in this section. First, we introduce the reader to decision models by means of a running example.

2.1 DMN

Decision Model and Notation (DMN) is a standard of the Object Management Group (OMG) to model operative decisions of enterprises. A decision model considers two layers: the requirements and the logic. The former provides a high level view on the information required for a decision, while the latter provides detailed information on how to take a decision [3].

For the remainder of this paper, we consider the following example: a manufacturer of bearings offers his customers reimbursement for delivered defective units. Therefore, a decision provides the fine as a ratio of the purchase price. In order to perform the decision, both manufacturer and customer must provide data. Figure 1 depicts the respective *Decision Requirements Diagram* (DRD), a graphical representation of the decisions and their dependencies. A node represents an entity, such as data (rounded shapes) and decisions (rectangles). The edges represent the information requirement relation indicating that data or decision results are required by a decision. The example consists of two decisions: *SLA* (service level agreement) and *fine*. SLA requires *years as customer* and *number of units* as data input. The decision *fine* requires the output of SLA and, additionally, the ratio of *defective units*. We marked the defective units in gray because it is contributed by the customer. The other (white) data inputs are provided by the manufacturer.

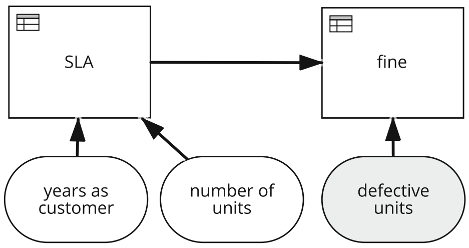


Fig. 1. Decision requirements diagram for the scenario; coloring of data input indicates the provider of the respective data, as detailed in the text.

On a second level, decision models specify the decision logic. DMN offers various notations, but we solely consider decision tables because they are most widely used. The decision tables that we use in this paper represent rules horizontally. A set of conditions on the data inputs is followed by specific decision outputs.

Table 1. Decision table for the SLA

U	*Inputs*		*Output*
	years as customer	number of units	SLA
	Number	*Number*	{*1,2* }
1	<2	<1000	1
2	<2	≥1000	2
3	≥ 2	< 500	1
4	≥ 2	≥ 500	2

Table 2. Decision table for the fine

U	*Inputs*		*Output*
	defective units	SLA	Fine
	Number	{*1,2* }	*Number*
1	<5%	1	0%
2	[5%..10%]	1	2%
3	>10%	1	100%
4	<1%	2	0%
5	[1%..5%]	2	5%
6	>5%	2	105%

Tables 1 and 2 depict the decision tables for our scenario. The table body consists of rules (i.e. four for SLA). In addition, the table header provides meta information. It specifies the names and data types of inputs and outputs: *years as customer* and *number of units* are numbers, the *SLA* is an enumeration with the possible values 1 and 2. The upper left cell of the decision table contains the *hit-policy*. The hit-policy determines in which order the rules of the table are evaluated. The used hit-policy *unique* (U) indicates that no two rules overlap in their conditions. Thus, at most one rule triggers for a specific input, and the evaluation order does not matter. More generally, we distinguish between single and multi-hit policies depending on the number of rules that contribute to the

Table 3. A description of DMN hit-policies

Sign	Name	Description
U	Unique	At most one decision matches a given input
F	First	The rules are evaluated in order. The first rule that matches the input determines the output
P	Priority	Output values must be enumerated and sorted according to priorities. If multiple rules match, the output with the highest priority is chosen
A	Any	If the condition of different rules overlap, the output must be the same. Thus, the evaluation order is irrelevant
C	Collect	All rules are evaluated and the outputs of all matching rules are collected. Afterwards an aggregation function (+, avg, min, max) is applied
R	Rule Order	The results of all matching rules are returned in the order of the rules
O	Output Order	Outputs are enumerated and ordered according to priorities. The outputs of all matching rules are provided in the order of their priority

decision's result (one vs. multiple). Table 3 provides an overview of the different hit policies. In this paper, we map all the elements of DMN decision models and decision tables in particular to Source code for Ethereum. This includes all elements mentioned above.

2.2 Blockchain Technology

Blockchain technology emerged with Bitcoin: a cryptocurrency whose transactions are stored on a blockchain [8]. Ever since, the technology has been adapted for different domains (e.g. BPM and logistics) and for various use-cases. However, all adaptations rely on a set of core properties [9].

A blockchain is a linked list that uses hash pointers. It is shared by a peer-to-peer network. Each element of the list is a block that stores transactions. The blockchain represents a history of transactions that allows us to restore all past and the present states. Some nodes, called *miners*, propose new blocks and find consensus about the set and order of transactions. Once this is achieved, all nodes share the same blockchain [10,11]. Thereby, *relative immutability* is reached: every node can detect inconsistent changes (e.g. changing the past) and reject respective transactions. However, if a majority of nodes agrees on a different version, the chain is adapted accordingly. The probability of a block being immutable increases as the number of succeeding blocks grows.

Another feature of (most) blockchains are cryptocurrencies. They are inherent currencies that are passed between users to reward mining, to pay for transactions, or to perform financial transactions. The so called 2$^{\text{nd}}$ generation of blockchains additionally supports complex *smart contracts*: Turing complete scripts, which are stored on the ledger and are executed by all miners. Thereby, immutability, publicity, and cryptocurrencies become available for computer programs. Smart contracts are like classes in an object-oriented language: they encapsulate data and functionality and can be instantiated.

Blockchains' properties are powerful since they enable interactions between participants without the need for trust. In recent research Mendling and others investigated the challenges and opportunities for BPM [12]. For one, process choreographies require trust between participants, but, as we have seen before, a blockchain can be a replacement by offering tamper-proofed monitoring capabilities. Implementations of process choreographies exist for both the Ethereum [6] and the Bitcoin [13] blockchain.

3 Generation of Decision Contracts from DMN

In general, contracts are agreements, about services, products, and money that participants agree to provide to each other. Decisions can be part of these agreements. In order to make a decision, the information has to be provided and logic needs to be executed. In an interactive process, each participant might contribute data to a single decision. Thus, participants need to exchange data to perform a certain decision. The DMN standard suggests to map each decision of a decision model to a decision taken by a single participant. Consequently, each participant will communicate the result of their decision such that others can base their decisions on this result. Figure 2 shows a respective business process collaboration for the running example: the manufacturer determines the SLA, the output of which is used as input by the customer to calculate the fine. In conclusion both decision results are exchanged.

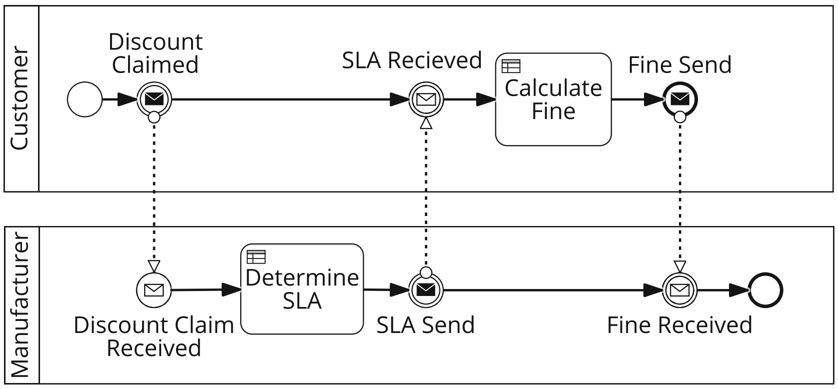


Fig. 2. DMN suggestion for decisions in interactive processes applied to the running example

This method of realizing decisions has a major drawback: due to the lack of a central mediator or state, participants must trust each other to provide the correct information and to take the decisions in the right way. In case of a fraud, conflict resolution is difficult: no accountable source of information exists and an expensive lawsuit can follow.

Weber et al. showed that a blockchain provides accountable audits and can be used as a trusted intermediary for interactive processes [6]. In the same fashion, we use blockchain as an intermediary for decisions. The respective process collaboration via blockchain is depicted in Fig. 3. We use Ethereum and smart contracts to specify the executable decision logic. Further, participants can contribute data to a decision via transactions, and smart contracts execute and log decisions. This leads to a public audit trail, which consists of all information required to reconstruct the decision making process: who provided which inputs, what are the results, and how was the decision made.

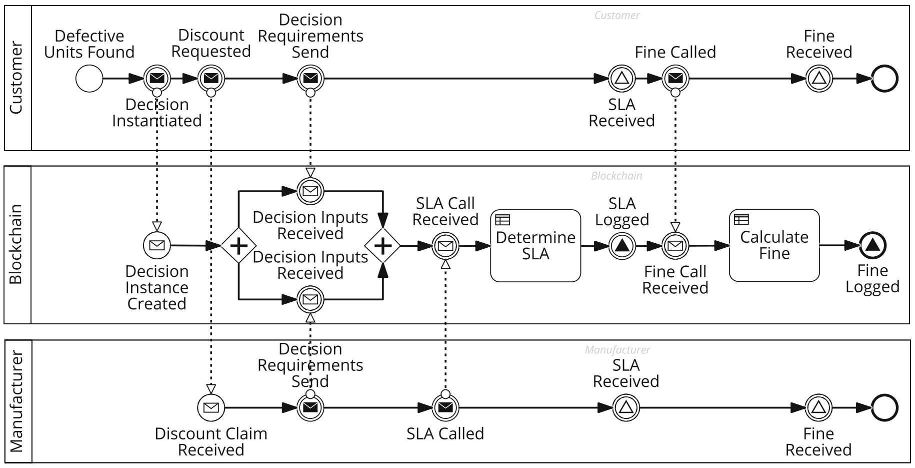


Fig. 3. Blockchain based implementation of a sample decision in interactive processes

Figure 4 depicts the mapping on the model level, where concrete decision tables are defined. As mentioned, a decision table defines the logic of a decision: it specifies inputs, outputs, rules, and a hit-policy. In our approach, we map each DMN decision table to a respective smart contract, which we call *decision contract*. A decision contract encapsulates the logic of the translated decision table and represents it in a function (called *decide*). Further, we provide two auxiliary structures: a *state contract* and a *factory contract*. The state contract encapsulates all inputs and outputs. The decision contract can, therefore, be implemented in a stateless manner following the best practices of DMN. Whenever a decision should be made, we provide a state contract instance with the respective data to the decision contract, which updates the outputs (inside the state contract) accordingly. The factory contract provides auxiliary functions

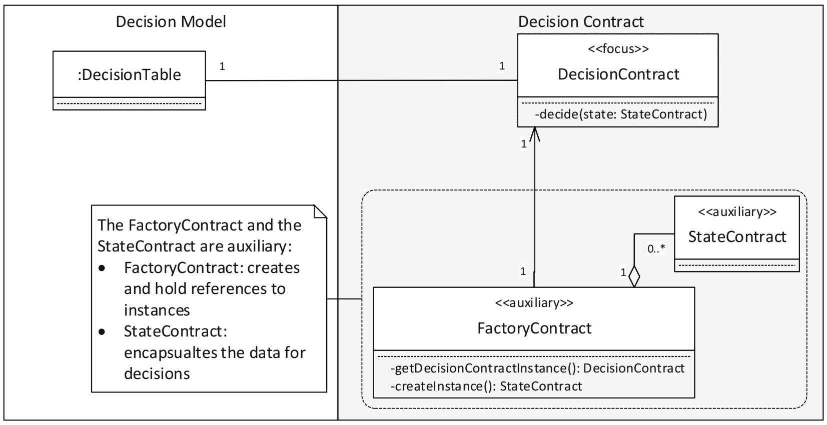

Decision Model
:DecisionTable
1
Decision Contract
<<focus>>
DecisionContract
-decide(state: StateContract)
1
1
<<auxiliary>>
StateContract
0..*
1
1
<<auxiliary>>
FactoryContract
-getDecisionContractInstance(): DecisionContract
-createInstance(): StateContract
The FactoryContract and the StateContract are auxiliary:
• FactoryContract: creates and hold references to instances
• StateContract: encapsualtes the data for decisions

Fig. 4. Mapping of DMN decisions (left) to Ethereum smart contracts (right), smart contracts are deployed on a blockchain

for instantiation: it creates and holds new state contract instances and lazily initializes all decision contracts of a respective decision model.

The DMN standard provides the FEEL expression language to describe execution semantics of decisions. A subset of this language is called S-FEEL and can be used to express decision tables. Such a decision table consists of inputs and outputs, whereby, outputs can be input for other decision tables. A rule of a S-FEEL decision table consists of unary tests for each input (statements that evaluate to true or false for a specific value) and an expression for each output.

Our approach provides a translation of all these elements to Solidity source code. Figure 5 shows the mapping of abstract S-FEEL elements to Solidity code templates. The code is refined as further elements of the decision table are mapped (e.g. for each model, we create one state contract, for each input and output clause we add a respective attribute with a setter method to it). A decision contract additionally has an *event*: a mechanism to broadcast information. It is used to publish the results of a decision. For each decision table, we create one decision contract and for each rule we create/extend an if-statement by adding the unary tests to its condition and the output expression to its consequence. However, different hit policies require different rule generations. We abstracted from this details in Fig. 5 but described them in Table 4.

3.1 Interaction with the Decision Contracts

In order to persist the participant's agreement on a blockchain, it is sufficient to deploy a single instance of the `FactoryContract` The `FactoryContract` is capable of instantiating decision contracts and the state contract and, thereby includes, the binary representation of the `StateContract` and the decision contracts (`SLADecisionContract`, `FineDecisionContract`). For the remainder we assume that a respective instance has been deployed.

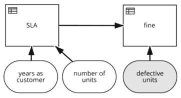
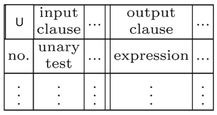
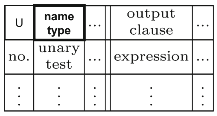
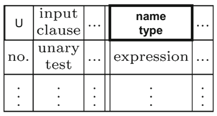
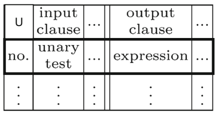
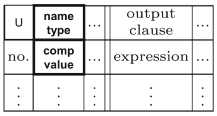
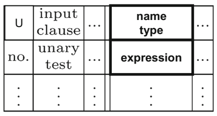
	DMN	Solidity
Decision model		`contract StateContract {[…]}`
Decision table		`contract DecisionNameContract {` `  event DecisionName(address instance, […]);` `  function decide(address _state) {` `    StateContract state =` `      StateContract(_state);` `    […]` `    DecisionName(_state, […]);` `}`
Input clause		`contract StateContract {` `  […]` `  type name;` `  function setName(type _value) {` `    name = _value;` `  }` `}` `contract DecisionNameContract {` `  function decide(address _state) {` `    […]` `    type name = state.name;` `    […]` `  }` `}`
Output clause		`contract StateContract {` `  // same as input clause` `}` `contract DecisionNameContract {` `  Event DecisionName([…], type name);` `  function decide(address _state) {` `    type name;` `    […]` `    DecisionName([…], name);` `    state.setName(name);` `  }` `}`
Rule		`function decide(address _state) {` `  […]` `  if (true […]) { […] }` `  […]` `}`
Unary test		`function decide(address _state) {` `  […]` `  if ([…] && name comp value) {` `    […]` `  }` `  […]` `}`
Expression		`function decide(address _state) {` `  […]` `  if ([…]) {` `    […]` `    name = expression;` `  }` `  […]` `}`

Fig. 5. Mapping of DMN elements to Solidity elements

Table 4. Handling of different hit-policies in `DecisionContracts`

Hit-policy	Description of mapping
Any, Unique, First	Condition is stored as an *if-else-if*; hence, the evaluation is complete after the first rule hit
Priority	Similar to Any, Unique, and First, but the rules are rearranged according to the output priority
Rule Order	Series of *if-statement* and output values are collected in arrays
Collect	Like Rule Order but an additional aggregation is applied in a postprocessing step
Output Order	Rules are rearranged in respect to the output order. Then, they are handled like Rule Order

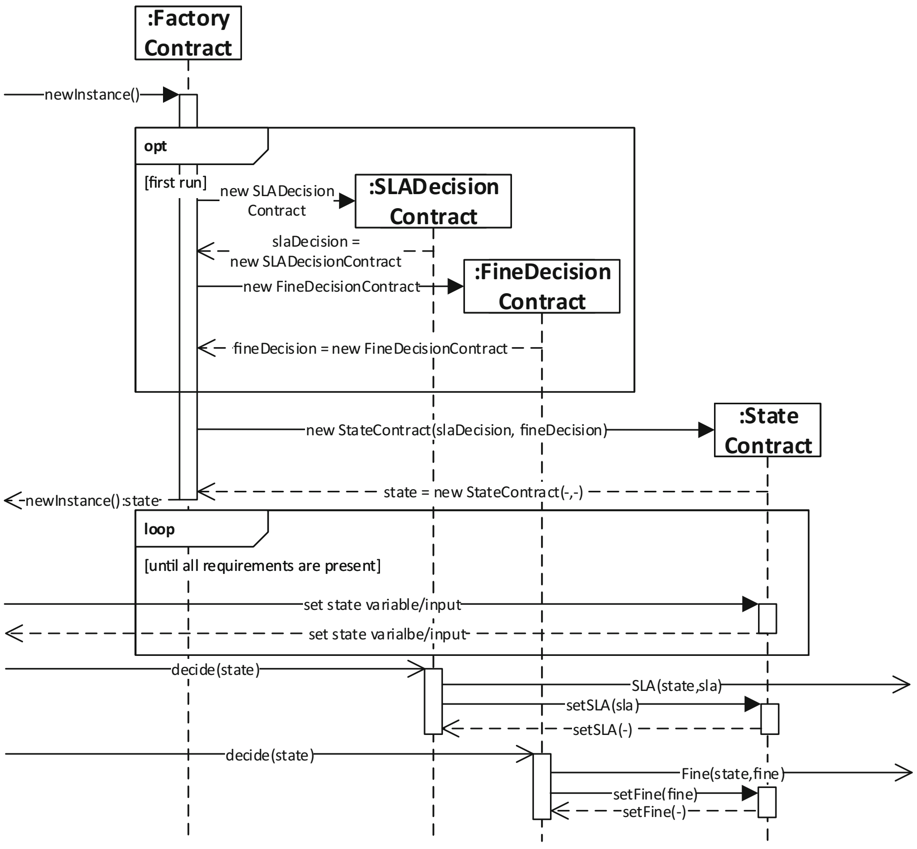


Fig. 6. Sequence diagram showing the interactions of the sample contracts

The sequence diagram in Fig. 6 shows the interaction of participants with smart contracts as well as among smart contracts. A participant (e.g. the manufacturer) calls the `FactoryContract` to create a new instance. If it has not been called before it initializes the decision contracts (i.e. `SLADecisionContract` and `FineDecisionContract`), and then creates a new `StateContract` instance with references to the decisions. The `StateContract`'s address is returned to the caller. Participants can then set input variables (i.e. years as customer, number of units, and defective units) on the `StateContract`, and thereby, provide the requirements for a decision. Once all decision requirements are fulfilled, a participant calls the decision(s) providing a `StateContract` instance. The decision logic is executed, outputs are determined, and eventually the state contract is updated. In this example, the output of SLA together with the defective units fulfills the requirements for the second decision, which can then be called.

4 Evaluation

To evaluate the approach, we investigate its applicability on the basis of a set of real life decision models. In addition, we provide an empirical cost evaluation of the example introduced above.

4.1 Applicability of the Mapping

To study the applicability of the mapping on real life decision models, we implemented the mapping in a proof-of-concept prototype. The prototype consists of a compiler, which translates DMN models (serialized as XML) to Solidity source code and is integrated into the Camunda modeler[1]. In this way, the Solidity source code is automatically derived from the decision model during design time.

Since Solidity is a Turing-complete language, S-FEEL can be translated to Solidity. However, due to Solidity's restricted type system, we encountered some engineering issues when defining the mapping.

S-FEEL's type *Number* is a generic floating-point number, but Solidity supports only integers and fractions. Therefore, floating-point numbers need to be represented with integers, which requires implementing auxiliary functions, for example, to compare two integer-based floating-point numbers. Another limitation comes from the fact that strings are stored as byte-arrays in Solidity, but only one-dimensional arrays can be dynamically sized. Therefore, lists of strings (requiring two-dimensional arrays) must be limited to items of a fixed length. Such lists are, for instance, returned by multi-hit decision tables.

We used the prototype to translate a real life set of decision tables to Solidity and from there to byte code for the Ethereum virtual machine. Our test data set consisted of 51 DMN models that were designed by participants of an online course on BPMN and DMN[2]. The prototype translated 46 of 51 models into

[1] https://camunda.org/download/modeler/.
[2] https://open.hpi.de/courses/bpm2016.

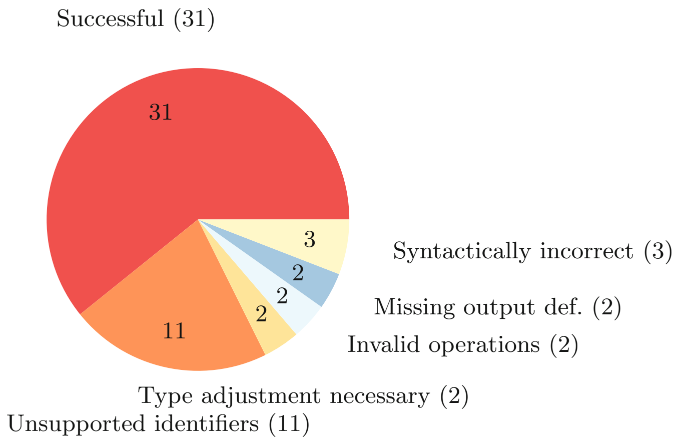


Fig. 7. Results of compiling the decision model test set to Solidity/Ethereum and reasons for failure

Solidity code. Of the remaining five, two models were missing an output definition (optional in DMN but required for our compiler) and three were syntactically incorrect. Afterwards, we ran the Solidity compiler to produce byte code for the Ethereum virtual machine. Out of the 46 remaining models 31 were compiled successfully. Two tables used long strings and compiled after adapting the data types manually. Eleven of the other 13 were not compiled, because identifiers for inputs or outputs used characters (e.g., brackets) or words (e.g., "contract") that are reserved words in Solidity. The remaining two decision models used comparison operators ($<, \leq, =, \geq, >$) on sets, which is forbidden in S-FEEL and in Solidity. These results of the investigation are summarized in Fig. 7.

To conclude, the majority of the decision models could be translated to executable Solidity code. Problems can occur because models are often used for documentation and not defined precisely enough for execution. Furthermore, special guidance during the modeling process could be helpful to prevent the use of reserved words or characters or from omitting necessary information.

4.2 Cost Evaluation

Ethereum is a second generation blockchain that integrates the full capabilities of a 1st generation blockchain—including a cryptocurrency called *Ether*. It is used to pay for transactions, i.e., all operations that store information in the blockchain. In our scenario, creating new contract instances, calling functions, and raising events causes transactions. In Ethereum, transaction costs are calculated in *gas*. Gas is an abstract unit that is mapped to Ether via a dynamic gas-price; in this way, the transaction costs can stay stable even though the value of Ether fluctuates.

The costs to instantiate smart contracts depend mainly on the required storage. Table 5 lists the cost for each contract after manually optimizing the used

Table 5. Instantiation costs for each contract and impact of features (based on avg. gas price and exchange rate on 11/07/2017 from https://etherscan.io/chart/gasprice)

Contract	Stripped	No optimization	Total
FactoryContract	1, 269, 518 gas 5.08€	153, 164 gas 0.61€	1, 422, 682 gas 5.69€
SLADecision	320, 558 gas 1.28€	41, 037 gas 0.16€	361, 595 gas 1.44€
FineDecision	494, 726 gas 1.98€	70, 869 gas 0.28€	565, 595 gas 2.26€
StateContract	172, 010 gas 0.69€	41, 233 gas 0.17€	213, 243 gas 0.86€
All	2, 256, 812 gas 9.03€	306, 303 gas 1.23€	2, 563, 115 gas 10.31€

data types and the respective savings. The `FactoryContract` is the most expensive one (5€) because it stores an array of instances, which can grow indefinitely and holds the definition of the other contracts to initialize them. The `FactoryContract` is for convenience: its functionality can be executed off-chain. This would save more than half of the instantiation costs. Moreover, we showed before that DMN data types cannot be mapped perfectly to Solidity types. In general, the latter supports more precise types, for example, integers of different sizes; thus, we can optimize them manually. However, this optimization has a relatively small impact on the costs saving only up to 1.23€.

All but the `StateContract` are initialized only once; if a decision is executed frequently, high instantiation costs can be tolerated. See Table 5 for details. The baseline costs of a single instance is the sum of instantiating the `State-Contract`, calling setters, and executing the decision logic (includes setters for the outcome). Figure 8 depicts the costs for a single instance. From bottom to top we sum the different operations. We show both a version with and a version without optimized types. It should come as no surprise that the instantiation is the most expensive operation. One can see that the costs of a single instance are about 1.70€ and type optimization saves up only 0.10€.

5 Related Work

Within the BPM community increasing interest is spent on blockchain technology. Mendling et al. give an overview of challenges and opportunities that arise from combining BPM and blockchains [12]. So far, existing research in this direction focuses mostly on business process choreographies. These efforts include an Ethereum based solution to enable and monitor choreographies [6]. All messages are logged and state progression is monitored via the blockchain. The current consensus on the blockchain describes the current state of the choreography and conform state progression is enforced. This concept has also been applied to the Bitcoin blockchain [13].

A similar approach is Caterpillar—a blockchain enabled process engine [14]. However, none of these approaches consider decisions and decision models. In Caterpillar XOR gateways are implemented by manually defining Solidity

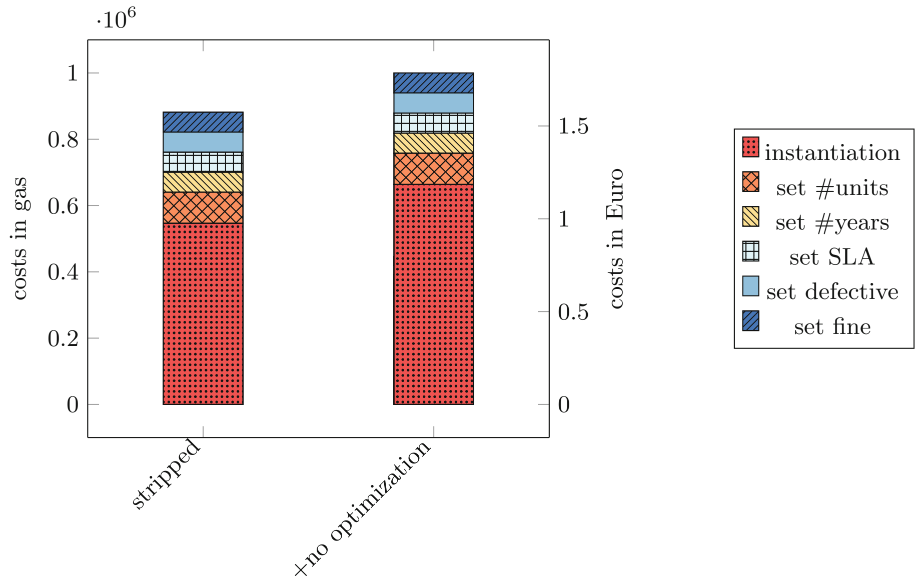


Fig. 8. Cost of an instance in *gas* and Euro—bottom to top: different operations, left to right: additional features

code snippets. Implementing XOR gateways for choreographies is a challenging task since a common understanding of both data and logic is required [2].

We propose smart contracts as a solution for implementing collaborative business decisions. The term smart contract has been formed by Szabo [15]. Kõlvart et al. summarize different smart contract developments and describe some opportunities of smart contracts on blockchains [16]. However, current smart contracts are implemented using procedural languages such as Solidity. In [17] Idelberger et al. discuss using logical programming languages instead because the respective contracts are more comprehensible and less error-prone. We build upon this idea by combining logical programming (decision tables) with a model based approach.

Most blockchains are public. This limits their usage to non-sensitive data [18]. The Ethereum community discusses this limitation and proposes *secret sharing decentralized autonomous* organizations that use data partitioning to sustain privacy [19]. Kosba and others take this idea further and present Hawk: a blockchain, which supports smart contracts and protects privacy through encryption [20]. Privacy protecting blockchains (supporting smart contracts), are not ready for productive usage, yet. Therefore, private and permissioned blockchains can be used to create a peer-to-peer network with a restricted set of nodes.

6 Conclusion and Discussion

Collaborative decisions span across multiple processes and organizations. They apply to interacting business process and influence their behavior. In these cases, it is difficult to establish a trusted exchange of data and a common understanding of both the information and the decision logic [2]. Therefore, we proposed the use of a decentralized blockchain as an intermediary entity. It saves both the data that is required for the decision and the underlying logic. Based on the blockchain technology, trust becomes obsolete due to a technical and reliable mechanism. We also showed a prototypical implementation that automatically translates DMN models into smart contracts. However, current limitations of the type system of Solidity (a language to write smart contracts) requires manual adjustment and workarounds, for example, to handle floating-point numbers.

To make our approach feasible, we extended the mapping with additional features: A push mechanism calls decisions automatically from the state contract. In addition, access rights limit the write access for each attribute to a set of participants or contracts (addresses); hence, unauthorized users cannot temper with the data. Further, only decision contracts can set their respective outputs.

An important issue of modern blockchains, such as Ethereum, is that all data is public. Everything saved to the ledger is visible to every node. This is not an issue for "public" contracts such as the terms of service of a company. However, sensitive or enterprise internal data cannot be stored on the blockchain. Alternatively, one can build a private or permissioned blockchain or use a new kind of blockchain such as Hawk [20] that offers built-in privacy mechanisms such as secret sharing decentralized autonomous organizations [19] and zero-knowledge proofs.

To conclude, DMN decisions can be implemented with blockchain technology making trust and a centralized third party, such as a DMN decision service, obsolete. Therefore, we closed the gap left open by previous work to execute interacting business processes on a blockchain by also taking the decision perspective into account.

Acknowledgements. We thank Alexander Kastius for his valuable contribution to the prototype implementation.

References

1. Weske, M.: Business Process Management - Concepts, Languages, Architectures, 2nd edn. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28616-2
2. OMG: Business process model and notation, specification 2.0. Version 2 (2011)
3. OMG: Decision model and notation, specification 1.1. Version 1.1 (2016)
4. Batoulis, K., Meyer, A., Bazhenova, E., Decker, G., Weske, M.: Extracting decision logic from process models. In: Zdravkovic, J., Kirikova, M., Johannesson, P. (eds.) CAiSE 2015. LNCS, vol. 9097, pp. 349–366. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19069-3_22

5. van der Aalst, W.M.P., Weske, M.: The P2P approach to interorganizational workflows. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 140–156. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45341-5_10
6. Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A., Mendling, J.: Untrusted business process monitoring and execution using blockchain. In: La Rosa, M., Loos, P., Pastor, O. (eds.) BPM 2016. LNCS, vol. 9850, pp. 329–347. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45348-4_19
7. Dannen, C.: Introducing Ethereum and Solidity. Apress, Berkeley (2017). https://doi.org/10.1007/978-1-4842-2535-6
8. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
9. Tschorsch, F., Scheuermann, B.: Bitcoin and beyond: a technical survey on decentralized digital currencies. IEEE Commun. Surv. Tutor. **18**(3), 2084–2123 (2016)
10. Narayanan, A., Bonneau, J., Felten, E.W., Miller, A., Goldfeder, S.: Bitcoin and Cryptocurrency Technologies - A Comprehensive Introduction. Princeton University Press, Princeton (2016)
11. de Kruijff, J., Weigand, H.: Understanding the blockchain using enterprise ontology. In: Dubois, E., Pohl, K. (eds.) CAiSE 2017. LNCS, vol. 10253, pp. 29–43. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59536-8_3
12. Mendling, J., Weber, I., et al.: Blockchains for business process management - challenges and opportunities. CoRR abs/1704.03610 (2017)
13. Prybila, C., Schulte, S., Hochreiner, C., Weber, I.: Runtime verification for business processes utilizing the bitcoin blockchain. CoRR abs/1706.04404 (2017)
14. López-Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I.: Caterpillar: a blockchain-based business process management system. In: Proceedings of the BPM Demo Track and BPM Dissertation Award Co-Located with 15th International Conference on Business Process Modeling (BPM 2017), Barcelona, Spain, 13 September 2017 (2017)
15. Szabo, N.: Formalizing and securing relationships on public networks. First Monday **2**(9) (1997)
16. Kõlvart, M., Poola, M., Rull, A.: Smart contracts. In: Kerikmäe, T., Rull, A. (eds.) The Future of Law and eTechnologies, pp. 133–147. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-26896-5_7
17. Idelberger, F., Governatori, G., Riveret, R., Sartor, G.: Evaluation of logic-based smart contracts for blockchain systems. In: Alferes, J.J.J., Bertossi, L., Governatori, G., Fodor, P., Roman, D. (eds.) RuleML 2016. LNCS, vol. 9718, pp. 167–183. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42019-6_11
18. Atzei, N., Bartoletti, M., Cimoli, T.: A Survey of Attacks on Ethereum Smart Contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
19. Buterin, V.: Secret sharing DAOs: the other crypto 2.0 (2014)
20. Kosba, A.E., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: the blockchain model of cryptography and privacy-preserving smart contracts. In: IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, 22–26 May 2016, pp. 839–858 (2016)