An introduction to Commitment Based Smart Contracts using ReactionRuleML

Joost de Kruijff, Hans Weigand

Tilburg University, P.O. Box 90153
5000 LE Tilburg, The Netherlands
j.c.dekruijff@uvt.nl, h.weigand@uvt.nl

Abstract.
Smart contracts gain rapid exposure since the inception of blockchain technology. Today's smart contracts are coded in non-mainstream procedural programming languages (e.g. Solidity for Ethereum), which lifts the requirement to draft enterprise ready smart contract to both a legal professional and a programmer instead of only the former. In search for a smart contract language that reduces the threshold to draft one, this conceptual paper elaborates how business logic can be converted to executable code for commitment-based smart contracts. Hereby, a contract is viewed as a set of reciprocal commitments. The smart contract ensures the automated execution of all or most of these commitments. In order to leverage its event processing capabilities, Reaction RuleML has been used to appropriately represent the elements and working of passive and active rules within a commitment based smart

Keywords: RuleML, Reaction RuleML, Blockchain, Commitment-Based Smart Contracts

Smart contracts combine protocols with user interfaces to formalize and secure relationships over computer networks. Objectives and principles for the design of these systems are derived from legal principles, economic theory and theories of reliable and secure protocols [1]. The concept of smart contracts emerged in the 1990s, but only gained exposure since the inception of blockchain technology. Today's smart contracts are coded using imperative programming languages, in mainstream programming languages (e.g. Javascript and Go for Tendermint) and non-mainstream procedural languages (e.g. Solidity for Ethereum) or. Representing contractual terms in code rather than natural language, could bring clarity and predictability to agreements, as a smart contract could then be tested against a set of material facts, allowing legal professionals on either side to know precisely how the contract would execute in every computationally-possible outcome [2].

In order to make smart contracts more familiar to ordinary internet users, it is important to minimize the threshold to read and write them, in particular for users from the legal practice who draft paper-based or electronic legal contracts (from now: conventional contracts). In this context, [2] has built an argument for logic-based smart contracts. In contrast to procedural languages, declarative languages, and logic-based languages in particular, strive for the ideal of programming by wish: a legal professional states what he or she wants, and the computer figures out how to achieve it [2]. Declarative programming therefore splits into two separate parts: methods for humans on how to write wishes, and algorithms for computers that fulfil them [3]. The consideration to use declarative languages for smart contracts has several objectives. (1) Current implementations of smart contracts unintentionally add a third party to the process; the requirement for a programmer or programming

knowledge to write a smart contract, which is in sharp contrast to the promise of smart contracts to remove intermediaries [4] and lower or nullify transaction costs altogether [5]. (2) In common legal practice, procedural and regulatory rules are written in natural language. Legal professionals are not expected to become programmers or vice versa. (3) It is unlikely that a legal professional can verify the legal validity of a smart contract coded by a programmer in a procedural language and it is therefore unlikely that large businesses will adopt smart contracts under these circumstances. Based on these objectives, we believe that there is need for a logical format that smart contract verification interpreters, that are immutable and live on the blockchain, can actually understand [6]. Several languages such as LKIF (knowledge interchange), SBVR (knowledge discovery), PENELOPE, ConDec (temporal knowledge), ContractLog, OWL-S2, eSML [7] and RuleML, have been proposed to facilitate this process for various useful purposes, but mostly from a generic knowledge engineering context and none for smart contracts in specific.

Our research objective is to make a theoretical contribution to the science of contract languages for (logic-based) smart contracts. Chopra's concept of business contracts as a bundle of commitments is applied using RuleML as a markup language for drafting smart contracts. We evaluate existing approaches using declarative rules [8] and introduce an alternative using reaction rules via the RuleML sublanguage Reaction RuleML, whereby a commitment is evaluated as an (economic) event. The practical goal of this paper is to set a future direction for commitment-based smart contracts by exploring to what extent business rules in natural language can be translated to code that is applicable to any blockchain platform.

This paper is structured as follows. First, we explore the elements of commitment-based smart contract and possible execution styles, followed by an introduction to Reaction RuleML, code examples per execution style and a conclusion.

Elements of a Commitment-Based Smart Contract

A contract is a legally binding or valid agreement between two or more parties. The main objective of a contract is to fulfill a certain goal and to safeguard against undesirable outcomes, together referred to as contract robustness [2]. Contracts that are not robust may lead to transaction costs, expensive conflict resolution, or even a collapse of a transaction as a whole.

It might be objected that commitments represent the positive actions to be performed by the actors only. What about prohibitions? For instance, a music customer is not allowed to copy the music he can download. In some cases, like the music copying, there may be technical means to make the prohibited action impossible, but there are also actions of agents that are not under control. In these cases, what the parties should commit to, is to take the consequences (e.g. paying a penalty). The prohibition – don't do A – is reformulated as a contingency commitment – IF <A> THEN <consequence>, where a transformation is described between deontic logic and dynamic logic [9]. In the example, the customer commits himself to be a penalty when he has made an illegal copy. Our claim is that all contract clauses can be similarly represented as commitments (validation of this claim is out of the scope of this paper).

In the context of blockchain, a commitment is equal to a future transaction, the robustness of their smart contract depends on how its commitments relate to the goals of the contract parties [2]. This definition

implies that a commitment-based smart contract should at least contain goals, commitments, conditions to execute and timing constraints.

As each contract goal consists of reciprocal commitments. In line with the economic exchange pattern [10], the duality between parties consists of rights and obligations. For every *obligation* that A owes to B, there is a balancing *right* from B to A. Commitments can both be financial (e.g. payment) or non-financial (e.g. promise to deliver a good) and may or may not happen under certain circumstances and a specified time.

Since there is no formal way to schedule transaction events via the blockchain protocol itself [11], smart contracts should be coded in such a way that they are schedulable, based on timing or other conditions. In order to mitigate this limitation, various solutions are presented in the smart contract community with regard to smart contracts execution method.

Method	**Explanation**	**Advantage**	**Disadvantage**
Lazy Execution (Must call to execute paradigm)	Transactions are initiated by an actor, either manually or scheduled	Low compute costs Lower complexity due to lack of nesting inside contract code	Less automation and leverage of the power smart contracts
Eager Execution	The smart contract polls for events in order to trigger reciprocal transactions	Fully automated Transactions provide security that they happen. Humans may forget to initialize a transaction when they should.	Full automation results in higher compute complexity, which equals transaction costs.

Figure 1. Smart contract execution modes

In this paper, we provide code examples of business rules within commitment based smart contracts using Reaction RuleML, for both execution methods, as they are both relevant in practice today. Besides the execution method, we have made other considerations that can be summarized as follows:

- Transaction events are atomic, meaning that they happen in total ('execute' in the technical sense) or not happen at all.
- The business logic (or rules of engagement) for economic settlement, is preferred to executed on-chain
- Each commitment that has time constraints consists of at least two smart contracts, since a second smart contract is created that takes care of the scheduled call of business logic in the 'main' smart contract
- Smart contract transactions for settlement are initialized by the smart contract itself, not by the parties involved, since the smart contracts protects the interests of all actors.

RuleML

RuleML is as markup language with the ability to express business rules as modular, stand-alone units. It can be extended and possesses the ability to resolve conflicts using priorities and override predicates [8]. RuleML adopts Java's class versus method naming convention by distinguishing

upper-case type tags from lower-case role tags, and uses the Datalog sublanguage of Horn logic as its kernel foundation.

Commitments can be simulated by both branches of the RuleML family of rules; transformative (declarative) rules and reactive rules. Derivation rules are believed to provide the most accurate conceptual representation of a contract. The main goal of a declarative rule is knowledge generation [12], whereby the validation of a premise will induce or justify a conclusion. Multiple rules can hereby be related to a single conclusion. [8] used declarative rules convert natural language contracts (in its existing form) into executable code using RuleML. Their approach aimed to mitigate RuleML's limitative support for reasoning on deontic concepts and its lack of ability to identify the behavior of roles in the contract and contract violations. According to this view, monitoring on contract performance deals with normative concepts like obligation, permission, prohibition (violation) and behavior. Since commitments do not follow traditional contract logic, we broadened our search for other rule structures within the RuleML family that provide better fit for the event driven nature of commitment based smart contracts.

Reaction rules are concerned with the invocation of actions in response to (complex) events and actionable situations [13]. Reaction RuleML is the quasi-standard for representing reaction rules. It is regarded as a user-friendly XML- serialized sublanguage of RuleML and acts as an interchange format for reactive rules and rule based event-processing languages. Reaction rules using Reaction RuleML typically implement forward-chaining operational semantics for Condition-Action rules where changing conditions trigger update actions, like IF/THEN/ELSE (derivative reasoning), IF/DO (production rules), ON/DO (trigger rules), ON/IF/DO (Event-Condition-Action or ECA) or a variation of ON/IF/DO and IF/THEN (Knowledge Representation or KR) [14].

Blockchain is considered to be a distributed database to transfer value or value derivatives (tokens). The introduction of smart contract functionality brought along functionality from the active database domain, like inserting and triggering. Nevertheless, blockchain is not limited to be an active distributed database either, since it has the capability to interact with (complex) events that consider time- and other constraints. Other (non-exhaustive) meta-model considerations for commitment based smart contract rules are summarized below:

- Since we consider commitments as social economic events that may contain other events (e.g. to pay) [15], the rules themselves should be event oriented.
- These events should be callable or detectable, which allows IF conditions to be specific for detected events only.
- The meta-model should allow multiple event definitions to be part of the same rule procedure, in order to process a contract goal as a bundle of reciprocal commitments.
- It is preferred to support smart contract wide states (e.g. deposit percentage) that apply to the entire smart contract, instead of defining them on run-time. This will maximize re-usability and consistency when contracts grow in complexity.
- Prohibitions or violations are not similarly handled compared to conventional contracts [16]. The prohibition – don't do A – is reformulated as a contingency commitment – IF A THEN <consequence> or ON A DO <consequence> in order to eliminate the need for complex and hard-to-maintain ELSE statements.

The remainder of this paper shows how lazy and eager rules within the commitment based contract can be defined. RuleML inhibits three execution styles for rules: *passive*, *active* and *reasoning*. Figure x shows how passive and active rules align with the smart contract execution styles.

Method Blockchain	**Method Reaction RuleML**	**Explanation**
Lazy	Passive	The smart contract 'passively' waits for incoming events
Eager	Active	The smart contract 'actively' polls/detects occurred events

Figure 2. Aligning smart contract execution and Reaction RuleML execution modes

Smart contract transactions are in essence text messages that are mined by network nodes. As a result, we depict transactions as <Messages> in Reaction RuleMl. The code examples provided fully adhere to the design considerations as presented in the previous section

Lazy execution

Passive reaction rules wait for incoming events in order to trigger an action. In this example, we assume an <event> that contains a <Message> that is sent to the smart contract. This message may (transaction) or may not (query) change the state of the smart contract (e.g. balance), a condition that can be used in the <body>. For example, it could be verified is there is a change to the state or if there is an outstanding obligation between the sender and the receiver. Once the condition is fulfilled, the smart contract triggers an automatic response message <action>, which sends a <Message> to the blockchain network to settle the balance.

```
<Reaction style="passive">
	<event>
		<Message mode="inbound">
			<oid><Var>PaymentID</Var></oid>
			<protocol>Protocol</protocol>
			<sender>SenderPublicKey</sender>
			<receiver>ContractPublicKey</receiver>
			<content>
				<Var>Amount</Var>
			</content>
		</Message>
	</event>
	<body>
		<Naf>
			<Atom>
				<Var>NoObligationLeft</Var>
				<Var>T</Var>
			</Atom>
		</Naf>
	</body>
	<action>
		<Assert>
			<Message mode="outbound">
				<oid><Var>PaymentID</Var></oid>
				<protocol></protocol>
				<sender>ContractPublicKey</sender>
				<receiver>
					NetworkNodes
				</receiver>
				<content>
					<Var>Balance</Var>
				</content>
			</Message>
		</Assert>
```

```
        </action>
</Reaction>
```

The same workflow holds for rights as well. The fact that a party has the ability to call (or schedule) execution when they want, gives a sense of control to stakeholders with regards to when actions are executed and transaction costs are incurred. Due to the lower compute costs involved with lazy execution, it is a popular way of designing smart contract rules.

Eager execution

Active reaction rules poll in order to verify whether or not an event has taken place by checking the changes to the state of the smart contract. This method does not require an additional smart contract that serves as the scheduler. All code can be part of one smart contract. In this example, we verify every hour through an <event>, whether or not a payment has been made that changed the state or settled an obligation from the sender to the receiver. Similar to lazy execution, once the condition is fulfilled, the smart contract triggers an automatic response message <action>, which sends a <Message> to the blockchain network to settle the balance.

```
<Reaction style="active">
        <event>
                <Reaction>
                        <event>
                                <Atom>
                                        <Rel>everyHour</Rel>
                                        <Var>T</Var>
                                </Atom>
                        </event>
                        <action>
                                <Atom>
                                        <Rel>detect</Rel>
                                        <Var
                                           type="paymentEvent"
                                           mode="-">Payment</Var>
                                        <Var>T</Var>
                                </Atom>
                        </action>
                </Reaction>
        </event>
        <body>
                <Naf>
                        <Atom>
                                <Rel>NoObligationLeft</Rel>
                                <Var>T</Var>
                        </Atom>
                </Naf>
        </body>
        <action>
                <Assert>
                        <Message mode="outbound">
                                <oid><Var>PaymentID</Var></oid>
                                <protocol></protocol>
                                <sender>ContractPublicKey</sender>
                                <receiver>
                                        NetworkNodes
                                </receiver>
                                <content>
                                        <Var>Balance</Var>
                                </content>
                        </Message>
                </Assert>
        </action>
</Reaction>
```

Conclusion

This conceptual paper examines rule definition for commitment-based smart contracts using Reaction RuleML as the declarative smart contract language. We have distinguished two execution modes for smart contracts: lazy execution and eager execution. A commitment based smart contract code example is provided for each execution mode in order to illustrate how rules can be coded. These code examples comply to non-exhaustive design considerations with regards to reliability and accountability. Since both methods are used in parallel in practice, the paper aimed to provide a sufficient introduction to these modes and its semantics.

The Reaction RuleML examples in this paper are a first step. An important next step for the development of commitment based smart contracts as a concept is to apply it in distinctive use-cases and implementation options, as well as the verification of input events, in particular when these events are outside the blockchain. In addition, the code examples could be simplified by extending Reaction RuleML with dedicated properties for commitment based smart contracts.

References

1. Szabo, N.: Formalizing and securing relationships on public networks. First Monday 2, no. 9, 1997
2. Chopra, A.K., Singh M.P., Oren, N., Miles, S., Luck, M., Modgil, S., Desai, N.: Analyzing Contract Robustness through a Model of Commitments, Agent-Oriented Software Engineering XI, LNCS 6788 pp 17-36, 2011
3. Umeda, M., Wolf, A., Bartenstein, O., Geske, U., Seipel, D., Takata, O.: Declarative Programming for Knowledge Management, 16th Int Conf. on Applications of Declarative Programming and Knowledge Management, INAP 2005, Fukuoka, Japan, Oct 22-24, 2005
4. Cheng, L., Saw, T.J., Sargeant, C.: Smart Contracts: Bridging the Gap Between Expectation and Reality, Oxford Law Opinion, 11 July 2016
5. Levy, K.E.C.: Book-Smart, Not Street-Smart:Blockchain-Based Smart Contracts and The Social Workings of Law, Engaging Science, Technology, and Society 3, 1-15, 2017
6. Lam, H.P., Hashmi, M., Scofield, B.: Enabling Reasoning with LegalRuleML, International Symposium on Rules and Rule Markup Languages for the Semantic Web, RuleML 2016: Rule Technologies. Research, Tools, and Applications pp 241-257
7. Norta, A.: Designing a Smart-Contract Application Layer for Transacting Decentralized Autonomous Organizations, International Conference on Advances in Computing and Data Science, 11-12 November, 2016
8. Governatori, G., Rotolo, A.: Modelling Contracts Using RuleML, Legal Knowledge and Information Systems. The Seventeenth Annual Conference, pp. 141-150, 2004
9. Wieringa, R. Meyer, J.-J, Weigand, H. Specifying dynamic and deontic integrity constraints, Data & Knowledge Engineering, Volume 4, Issue 2, 1989, Pages 157-189
10. Blums, I., Weigand, H., Towards a Reference Ontology of Complex Economic Exchanges for Accounting Information Systems. EDOC 2016: 119-128.
11. StackOverflow, Januari 2016: https://ethereum.stackexchange.com/questions/42/how-can-a-contract-run-itself-at-a-later-time#87
12. Zhao, Z., Teymourian, K., Paschke, A., Boley, H., Athan, T.: Loosely-Coupled and Event-Messaged Interactions with Reaction RuleML 1.0 in Rule Responder, CEUR Workshop Proceedings, Vol. 874, Article 13, 2012

13. Paschke, A., Boley, H., Zhao, Z., Athan, T. : Reaction RuleML 1.0: Standardized Semantic Reaction Rules, Complex Reactivity with Preferences in Rule-Based Agents (pp.100-119), 2012
14. Paschke, A.: ECA-RuleML: An Approach combining ECA Rules with temporal interval-based KR Event/Action Logics and Transactional Update Logics, ECA-RuleML Proposal for "RuleML Reaction Rules Technical Goup", 2005.
15. De Kruijff, J.., Weigand, H,: Understanding the Blockchain Using Enterprise Ontology, 29th Int. Conference on Advanced Information Systems Engineering, 12-16 June 2017
16. De Kruijff, J., Weigand, H,: On the Move to Meaningful Internet Systems. OTM 2017 Conferences: Confederated International Conferences: CoopIS, C&TC, and ODBASE 2017, Rhodes, Greece, October 23-27, 2017, Proceedings, Part II (pp.383-398)