Self-Aware Smart Contracts with Legal Relevance

Alex Norta
Large-Scale-Systems Group, Department of Software Systems,
Tallinn University of Technology, 19086, Tallinn, Estonia
Email: alex.norta.phd@ieee.org

***Abstract*—Conventional contracts (CC) that are not machine readable are challenging to interpret, disconnected from ICT-systems and when conflicts occur, tracking their execution is restrictively slow and challenging to enforce. On the other hand, so called self-aware contracts (SAC) that are similar to CCs with respect to legal enforceability, are machine readable and supportable by blockchain-technology in combination with multi-agent systems. SACs do not require qualitative trust between contracting parties because blockchains establish instead a quantitative notion of trust as SAC-related events are immutably stored while software agents manage the connections to trusted information sources. However, currently existing machine-readable contract solutions, i.e., smart contracts, lack suitable obligation- and rights constructs for execution and enforcement. Additionally, current systems do not comprehend the dynamics of legal relationships. This whitepaper address the gap by specifying a so-called SAC-framework that enables blockchain-driven self-aware agents-assisted contracts for a decentralized peer-to-peer (P2P) economy.**

***Index Terms*—self aware, multi agent, blockchain, smart contract, decentralized, per-to-peer, e-governance**

I. Introduction

The traditional understanding of a conventional contract (CC) is an exchange of commitments by identified parties that are enforceable by law. An important prerequisite for a contract that most commonly exists as a written document of evidence, is the voluntary engagement of parties involved to establish a consensus [6]. In most business cases, CCs are documents [19] that identify the contracting parties uniquely and state explicitly the commitments of the latter. When those commitments are performed, their status changes over time. Another problem with the traditional form of setting up and managing CCs is that they are often underspecified and the ability to manually track their status is restricted. As there is no concrete overview of the CC-status, the contractual relationship between parties is prone to conflicts. The resulting costly conflict resolutions may even collapse an entire contractual relationship. Also the enforcement of CCs [13] proves to be either too complicated, time consuming, or impossible, certainly in international circumstances.

The authors in [7] recognize shared blockchain technology enables business collaborations that require high-reliability and shared, trusted, privacy-preserving, immutable data repositories for smart contracts. So-called business artifacts for adopting data-aware processes provide a basis on shared blockchains that enable business-collaboration languages such a Solidity [10] of Ethereum. In [23], the authors map a running case of a collaborative process onto a smart-contract scripting language. That approach addresses the trust-issue in collaborative processes in that no single third-party entity must monitor events. Instead, the blockchain enables trustless process collaboration because of no single entity being in control. The mapping from collaborative processes to blockchains enables the monitoring of process enactment and an auditing of related events. In [8], different smart-contract language choices are compared. While procedural languages are currently the norm[10], also logic-based languages are alternatives.

The state of the art above shows that partial smart-contract approaches exist for blockchain technology. However, there is a lack of a framework moving smart- towards self-aware contracts (SAC) where the latter have the ability to gather information about their internal and external-contextual state and progress to reason about their behavior, while being an artifact of law. Furthermore, the above described state of the art, there is not recognition that such SACs must cater for having humans in the contract loop. This paper fills the gap by posing the question how to establish self-aware and human-readable contracts as legally viable? To reduce complexity and establish a separation of concerns, we deduce three further sub-questions as follows. What enables contracts to be self aware? What processing mechanisms are required for a SAC? What means create a self-aware contract-exchange protocol?

Based on a pre-existing ICO[1] whitepaper [18], the remainder of this paper is structured as follows. Section II presents essential preliminaries. Section III focuses on a smart-contract ontology and Section IV discusses the processing of legal obligations and rights. Section V shows the role of software agents that allow for self-aware contract-collaboration protocols. Section VI evaluates the results of this research and finally, Section VII concludes the paper together with a future-work presentation.

II. Preliminaries

Scholarly literature about smart contracts exists in the legal domain. In [1], the core elements of legislation are

[1] https://www.agrello.org/

978-1-5090-6014-6/18/$31.00 ©2018 IEEE

addressed, including duties and obligations that share intersecting properties. The characteristic of a duty is the absence of a benefiting party (beneficiary), while the performance of an obligation serves a beneficial result for a determined beneficiary. The focus of the paper is on obligations the properties of which Figure 1 informally depicts.

The properties in Figure 1 show a micro-process for obligations development using the business-process modeling notation BPMN [12]. The green-lined circle denotes the start of the process and the red-lined circle the end. Rectangles in Figure 1 are tasks and x-labeled diamonds denote an exclusive-choice split and -join respectively. Directed arcs connect the nodes along a control flow from start to end. Figure 1 shows that obligations exist to either do something, or to refrain from something.

In contract law, rights and obligations are related so that if one party to the contract decides to use his right, there is a corresponding obligation on the other party. Thus, rights that stem from the contract are reflected in obligations of the other party. Figure 2 depicts a micro-lifecycle of rights specifications. After determining the beneficiary of a right, there can either be a right to claim, or a right to do something that pertains to an action type and object. Finally, the obligors must be determined who enable a right. For example, the lessee has a payment obligation in a rental contract. In case of a late payment, the lessor has the right to claim late-payment charges. After invoking that right, the lessee has an obligation to pay.

III. SAC Ontology

A SAC must comprise important elements of contracts to provide metadata during the contract execution. This metadata can then be used in various ways by informatics systems, but most importantly agents that assist, automate and manage contract execution. As mentioned above, rights and obligations must be optimized for machine readability. We show machine-readability in the sequel for rights and obligations while maintaining the capability for non-technical persons to comprehend the smart rights and obligations based SAC.

The ontology for this paper we design with the Protégé tool [14] that is a free, open source ontology editor for systematic knowledge acquisition. Protégé comprises a graphic user interface with plugins for varying ontology visualizations and correctness checks. We employ the HermiT reasoner [5] to check the ontology consistency, identify subsumption relationships between classes, and so on.

Figure 3 depicts the class diagram of the SAC-framework ontology[2]. Several sub-class relationships exist to capture all essential contractual elements. For example, we refine an obligation by adding as subclasses *Monetary_Obligation* and *NonMonetary_Obligation* to express certain remedies are only available for non-monetary obligations that can be a repair, or a replacement, while some are monetary, e.g., late-payment charges. There exist also person subclasses such as an *Obligor* who must perform an obligation, a *Beneficiary* who is benefiting from the performance of an obligation and optionally, a *Third_Party* as a beneficiary from the performance of an obligation, e.g., a utilities provider in a rental contract.

The purpose of the *Remedy* subclasses in Figure 3 is to eliminate negative consequences that result from a breach of the contract. Additionally, by invoking remedies, a beneficiary achieves a specific situation, if the obligation had been performed correctly. For example, if a rental payment is delayed for commercial property, the lessor can claim late-payment *Interest*.

The *Right* subclasses are important as they reflect what a *Beneficiary* can claim. For example, if a lessee destroys furniture in an apartment, the lessor has the right to *Claim_Repair*, or *Claim_Replacement*. Finally, the *State* subclasses in Figure 3 reflect the status of an *Obligation* performance in a contract lifecycle. We refer the reader to [18] for more details.

IV. Obligation Processing

Since the obligation ontology is static, we employ Coloured Petri Nets (CPN) [9] as a graphical oriented language for covering the dynamic aspects of obligation processing using CPNTools[3]. Informally, the CPN-notation comprises states, denoted as circles, transitions, denoted as rectangles, arcs that connect states and transitions but never states with other states or transitions with other transitions, and tokens with color, i.e., attributes with values. Arcs carry inscriptions in CPN-ML expressions that evaluate to a multiset or a single element. Modules in CPN are non-atomic place-holder nodes for hierarchic refinements that correspond to respective services in a system-implementation.

During a contract lifecycle, obligations move through stages of processing. According to the ontology classes of Figure 3, those stages are *inactive*, *active*, *performed*, *delayed*, *defective* and *terminated*. Additionally, there exist the stages *revised* and *unfulfillable*, which is out of focus for the automation of obligation processing. More precisely, we discuss the respective stages below:

- inactive: When an agent has not taken an obligation into consideration, i.e., the precondition of an obligation has not been met.
- active: An agent takes an obligation into consideration, i.e., the precondition of an obligation is met. That infers an obligor has to perform the related action before the deadline passes.
- performed: The action has been carried out by the obligor.
- delayed: The obligor has not carried out the action before the agreed deadline. Delayed state presumes that the amount of the action object in the obligation

[2]SAC-OWL: http://tinyurl.com/lkkapvg

[3]http://cpntools.org/

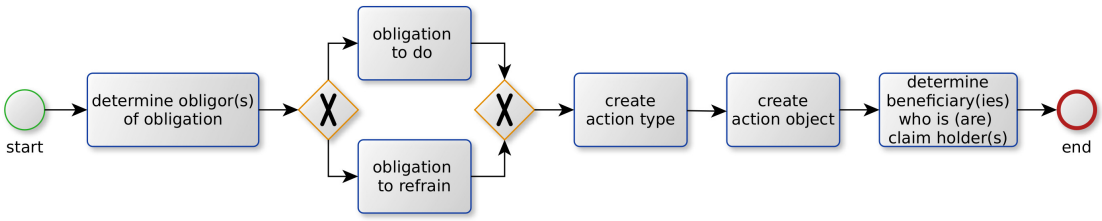

Fig. 1. Informal properties of an obligation.

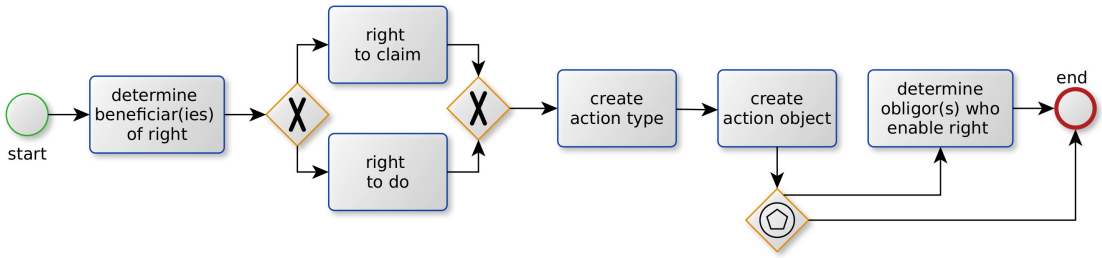

Fig. 2. Right-development micro-lifecycle.

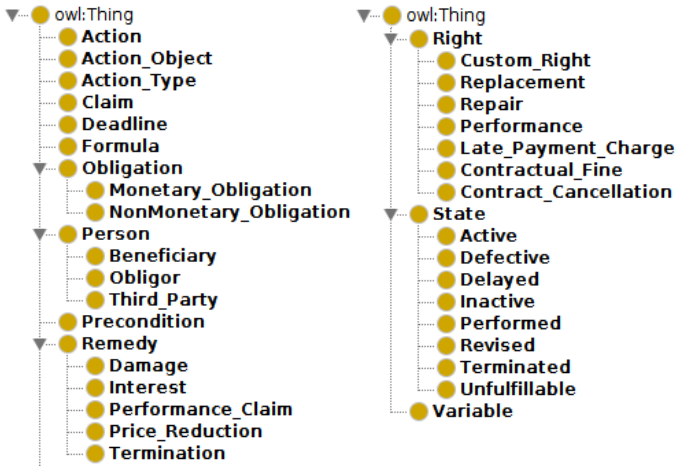

Fig. 3. SAC-ontology class diagram.

is not delivered to the beneficiary, or is not delivered in the sufficient amount.

- defective: The action object of an obligation is defective.
- terminated: The obligation can be terminated by a fundamental breach, or by mutual agreement. No further consideration of the obligation will take place.

Following the CPN model in Figure 4, when an obligation is in the stages *delayed*, or *defective*, a contractual agent starts reasoning about breaches to notify a collaborating party about the rights to remedy breaches, or other options for conflict resolution. In the *delayed* stage, the action object of the obligation is not delivered before the deadline passes, or is not delivered in the sufficient amount. For example the rent is not paid, or is paid less than required.

A *defective* distinction in Figure 4 shows monetary and non-monetary obligations. A monetary obligation includes a monetary action, while non-monetary obligation includes an action with a non-monetary action object. For example, the obligation to pay rent is a monetary obligation and the obligation to transfer the possession of an apartment is a non-monetary obligation. Only a non-monetary obligation can enter into a defective obligation stage. The latter requires the action object to lack the expected quality compared to agreement. For example when the lessee returns the possession of the apartment to the lessor without the apartment being in the agreed condition. In contrary to that, an obligation to pay rent cannot have qualitative deficiencies, because rent as the action object of the obligation has only quantitative features and does not have any qualitative ones. Although being in the state *performed*, the obligation can go to the state *defective* if defects are discovered in the aftermath.

The obligation stages *delayed* and *defective* in Figure 4 initiate rights to the beneficiary of an obligation to claim remedies. The *delayed* stage can initiate rights to claim performance, late-payment charges for monetary obligations and a contractual fine for non-monetary obligations. The *defective* stage can only be reached by non-monitory obligations and it allows the beneficiary to claim repair, or replacement while also being able to claim damages.

When the remedies in Figure 4 do not enable the beneficiary to achieve the purpose of the obligation perfor-

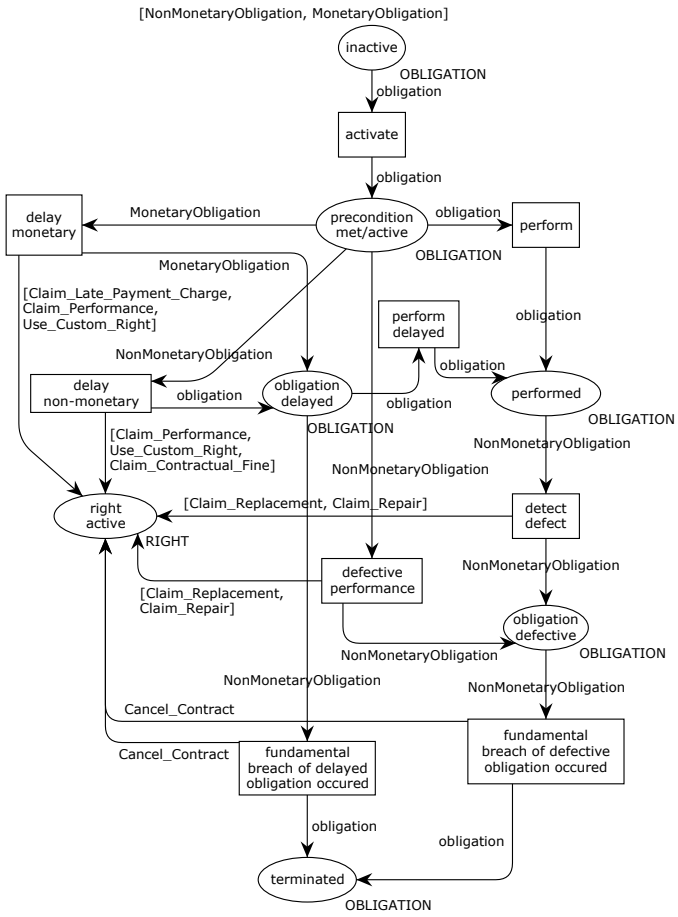

Fig. 4. Transaction processing of obligations.

mance, the obligation is fundamentally breached, resulting in the obligation reaching the stage of *terminated*. This can initiate the right for the beneficiary to cancel the contract. The obligation can also be put into the stage *terminated* at any time by the mutual agreement of the parties.

V. Interacting Contract Agents

The need emerges for employing SACs where smart contracts are combined with smart objects such as Belief-Desire-Intention (BDI) agents [2]. Jason[4] is a platform for the development of BDI-agent systems that incorporates a reasoning cycle for interpreting and executing source code in the agent-oriented programming language AgentSpeak. The latter stems from logics programming and allows for knowledge presentation in mathematical relations.

In Figure 5, we use a UML sequence diagram [20] to show the SAC-automated establishment of a rental contract in a smart housing setting for which we refer the reader to [18] for details. Thus, in Figure 5 we assign a fictitious public key number that comprises four characters for readability. The three entities to the left represent a contractual agent and two personal agents for the lessor and lessee respectively. The fourth entity denotes the blockchain into which events are registered. Furthermore, we assume a smart-home scenario where the apartment has four agents assigned, one for the smart lock and three for the gas-, water- and electricity-smart meters respectively.

The sequence diagram in Figure 5 commences with the contract agent sending a message to the lessee agent about the obligation $ob(dep)$ being active, meaning that the deposit must be paid. Consequently, the lessee agent performs the payment by invoking $tx(dep, 7a30)$ to the blockchain, i.e., the deposit is held by the contract agent. Note that usually a deposit is paid to a lessor's account, which is problematic as the lessor has the exclusive control over funds that he is not entitled to unless there is damage done to the apartment. At times, the deposit is never paid back to the lessee, even when the apartment is in undamaged condition. Still, in the case of a contract-agent wallet, the parties are forced to find a consensus about the deposit.

The next message in Figure 5 is from the contract agent to the blockchain for checking the unspent transaction output $UTXO(7a30)$ to assure the deposit is transferred to the contract-agent wallet onto the blockchain. The latter responds with a confirmation – *result(bitcoinC 0.3)* – indicating the payment to the contract-agent's public-key address on the blockchain.

For the formation of the transfer-act of the smart property, several types of information must be collected. First, the contract agent sends an active obligation message *ob(pt)* to the lessor agent for requesting pictures of the apartment condition. The lessor agent responds by delivering those pictures. Next, value-query messages *qu(value)* are sent by the contract agent to the gas-, water- and electricity agents respectively, who respond with sending back the current smart-meter counts in *result(value)*-messages. The latter are used by the contract agent in combination with the pictures to generate the transfer act that is subsequently recorded in the blockchain.

Next, the contract agent sends an active obligation message *ob(key)* to the lessor agent who subsequently sends another message *order(transfer,31x7)* to the smart-key agent, i.e., the smart key to the apartment is now usable by the lessee. Note that by using the blockchain for smart-key assignment, it is possible to perform an assignment to multiple persons and the lessor is aware of their identity. Finally, the contract agent sends an active obligation message *ob(rentpay)* to the lessee agent, after which the latter sends a transaction *tx(rent,03m6)* to the blockchain, i.e., the recipient of the first monthly rent payment is the lessor.

The termination protocol for the apartment rental contract in Figure 6 commences with the contract agent sending an active obligation message *ob(pic)* to the lessee agent who returns a set of pictures about the apartment conditions. Next, the smart-meter values are requested with

[4]http://jason.sourceforge.net/wp/

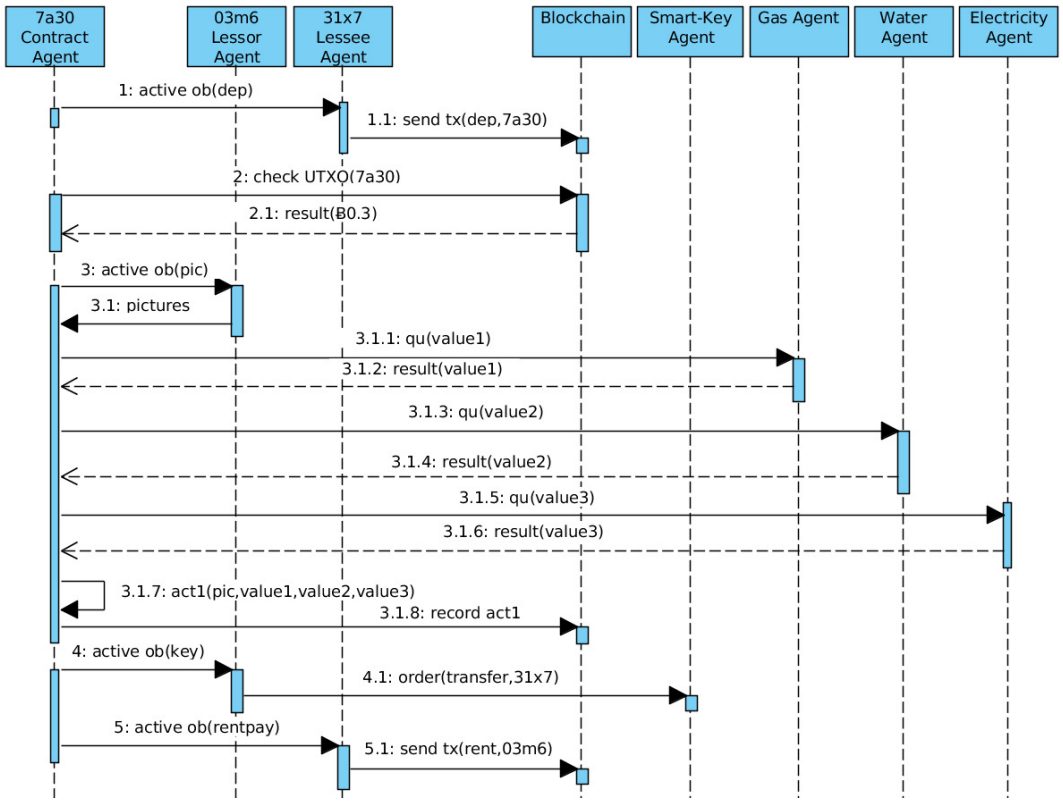

Fig. 5. Initiation protocol of contractual agents.

qu(value)-messages from the gas-, water- and electricity agents respectively. The latter respond with *result(value)*-messages from the respective smart meters. Assuming the delivered pictures about the apartment condition are accepted by the lessor, the contract agent invokes the command *act2(pic,value1,value2,value3)*. The generated *act2* is recorder into the blockchain and the contract agent sends an active obligation message *ob(key)* to the lessee agent, indicating the apartment smart key must be returned to the lessor. Consequently, the lessee agent sends a message *order(transfer,03m6)* to the smart-key agent.

The contract agent informs the lessor agent with the message *right(damage claim)* that there should be a final confirming check for possible damage compensation. We assume in Figure 6 that no damage compensation occurs and subsequently, the contract agent sends an active obligation message *ob(dep)* to the lessor for indicating the deposit must be paid back to the lessee. For that, the lessor agent sends a transaction message *tx(dep,31x7)* to the blockchain. Finally, the contract agent sends a check command *UTXO(7a30)* to the blockchain, after which the latter responds with the message *result(B0.3)*, i.e., the deposit has successfully been returned to the lessee.

VI. Feasibility Evaluation

We show the high-level structure of the business-collaboration language we call SAC-Language that is derived from the research-driven and pre-existing eSourcing Markup Language (eSML) [17] schema as a foundation. SAC-Language is given in Extensible Markup Language (XML) [3] to facilitate the building of distributed applications in Clouds [4].

Figure 7 shows the structure of the SAC-Language as a SAC between collaborating parties, structuring the SAC-Language content into the conceptual blocks Who, Where, and What. Briefly, the *Who* block comprises constructs for the resource definition and the data definition. Mapped onto the running apartment-renting case, parts of the resource definition are the housing company, the utility smart meters, and related information. Note that BDI-agents count as resources and are therefore defined with an unique identifier and universal resource identifier (URI) [11].

The *Where* block defines the business context in terms of used business, legal, and geographical aspects are of importance for the contractual relations of collaborating parties. In the context of the renting case, we assume Estonian jurisdiction holds. More concretely, the business-context provisions comprise obligations and rights that are assigned to concrete process tasks we explain below. The legal context provisions allow for setting general terms and conditions for a contract.

In the *What* block, the current adoption of a formal process-specification language permits the use of control-

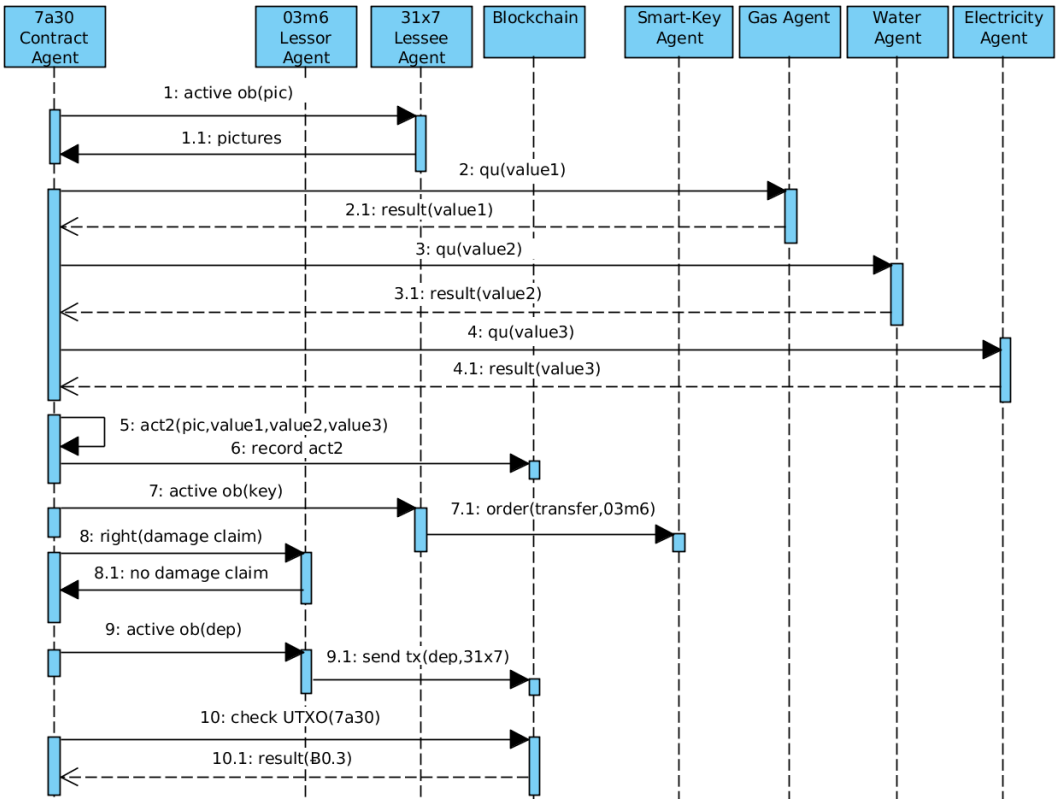

Fig. 6. Termination protocol of contractual agents.

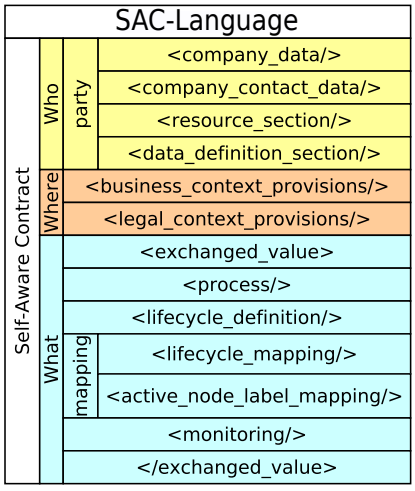

Fig. 7. SAC-language structure.

flow patterns for business-process definitions that have semantic clarity [16]. Note that the process definitions comprises constructs for linking to the resource- and data-definition sections of SAC-Language that are both based on respective pattern collections [22], [21]. Furthermore, life-cycle definitions [15] are for the business processes and contained tasks.

Since the SAC framework is blockchain agnostic, the mapping assures that heterogeneous organizational-internal smart-contract platforms can be integrated cross-organizationally. Thus, the life-cycle-mapping establishes semantic equivalence between the life-cycles of the cross-organizationally harmonized business processes and of tasks from the opposing domains. Different labels of tasks belonging to processes of opposing domains may be semantically equal. To establish a semantic equality, the second part of the mapping block focuses on the mapping of task labels. The monitoring construct of Figure 7 specifies how much of the enactment phase the service consumer perceives. We refer the reader to [15] for further details.

For the running rental case, we give brief SAC-Language code examples for obligations and rights. Listing 1 shows an example for the obligation to pay monthly rent. We assume the obligation has a name and unique ID, can not be changed throughout the enactment of a SAC and involves monetary units for the execution.

The state of the obligation in Listing 1 is *enabled*, i.e., the SAC enactment is at a lifecycle stage where the obligation is active. Next, the parties of the obligations define as a beneficiary the lessor. Note, we use the shrunk public key number of the lessor wallet from Figure 5. The same holds for the lessee who is defined as the obligor and must pay the monthly rent. There is no third party involved in this obligation. Following Figure 1, the obligation type is

todo in that the lessee has to act by concretely paying the rent.

Listing 1. Obligation example for paying monthly rent.

```
<obligation_rule tag_name="monthly_rent"
rule_id="0001" changeable="false"
monetary="true">
<state>enabled</state>
<parties>
<benficiary>Lessor(31x7)</beneficiary>
<obligor>Lessee(03m6)</obligor>
<third_party>nil</third_party>
</parties>
<obligation_type>todo</obligation_type>
<precondition>
act1(signed)&key(transferred)
<precondition>
<action_type>
payment(03m6,31x7,rent)
</action_type>
<action_object>
rent(monthly,amount)
<action_object>
<rule_conditions>
month(lastday)
</rule_conditions>
<remedy>
late_payment_interest(amount,03m6,31x7)
</remedy>
</obligation_rule>
```

As a precondition for the obligation in Listing 1, the *act*1 must be signed by the lessor and lessee while the latter must have access to the smart key for being able to move into the apartment. The action type is the payment from the wallet of the lessee to the lessor that constitutes the type *rent*. Additionally, and conforming to Figure 1, the action object is defined as the rent with the qualifiers it must be serviced monthly for a specific amount. The rule condition is that the rent payment must occur on the last day of a month. Finally, a reference is inserted in the obligation that a remedy for late rent payment exists where the lessee must transfer a defined monetary amount to the lessor.

The right in Listing 2 comprises intersecting specification elements with an obligation. As pointed out in Section II, the main difference with an obligation is the the beneficiary may waive a right. We assume in the right example of Listing 2 the hypothetical case the lessee has broken a television for which the lessor is the owner.

The right is again defined by a corresponding name and ID. As the lessor has the right to waive the right e.g., in case the lessee convinces the lessor the television damage is not her fault even when no evidence exists, the right can be changed on the fly and the compensation is set to $false$ as the expectation is a full replacement of the object. The right is in the lifecycle state *enabled* for immediate enactment and the parties are similarly defined as in Listing 1.

Corresponding to Figure 2, the type of the right is set to *claim* pertaining to the lessor over the lessee for a replacement of the broken television. The assumed precondition is again that *act*1 is signed and the smart-key handover to the lessee took place. The action type is a replacement of the television that is defined as an object by *brand*,*type* and *serial_number*.

Listing 2. Right example for replacing a broken televsion.

```
<right_rule tag_name="TV_replacement"
rule_id="0002" changeable="true"
monetary="false">
<state>enabled</state>
<parties>
<benficiary>Lessor(31x7)</beneficiary>
<obligor>Lessee(03m6)</obligor>
<third_party>nil</third_party>
</parties>
<right_type>claim</right_type>
<precondition>
act1(signed)&key(transferred)
<precondition>
<action_type>
replace(tv)
</action_type>
<action_object>
tv(brand,type,serial_number)
<action_object>
<rule_conditions>
deadline(date)
</rule_conditions>
<remedy>
late_replacement_interest(amount,31x7)
</remedy>
</right_rule>
```

We assume that the *replace*() command must be confirmed via mobile phone by the lessee with a photo showing the television being delivered to the mobile phone of the lessor. The obligation in Listing 2 also has a certain *date* set as a deadline for the television replacement. Otherwise, the lessee must again service a remedy payment of a certain amount to the wallet of the lessor.

VII. Conclusion

This whitepaper presents a novel cross-organizational blockchain-agnostic framework for peer-to-peer collaboration. Novel blockchain technology enabled smart contracts, combined with intelligent multi-agent systems yield so-called self-aware contracts that allow for a high degree of automation for such peer-to-peer collaborations. Since existing blockchain-based solutions lack essential constructs for specifying legally binding, machine-readable contracts, we pragmatically formalize obligations and rights with an ontology. For processing obligations and rights, a high-level state-transition automata in Colored Petri Nets shows the processing semantics involving a blockchain that assures event traceability. Important is that the self-aware contracting language constitutes a high-level, cross-organizational, declarative way of formulating self-aware contracts that are human readable and comprise specifications of obligations and rights, which are mapped onto organization-internal smart-contract transaction-processing platforms.

We discover that the combination of belief-desire-intention agents together with the declarative SAC-Language yields self-aware contracts where the former

assure as a combined set trusted information is channeled into contract-based collaborations. That way, agents create a composed oracle. In addition to employing agents that provide a degree of artificial intelligence in a collaboration, human manageability of the self-aware contract framework we achieve by providing a declarative smart-contract language that specifies cross-organizational contract-collaborations. This SEC-Language is based on a pre-existing language that results from an EU-project for initially automating cross-organizational production processes. The SEC-Language provides extensions by adopting human-readable specifications for obligations and rights, which are core concepts for lawyers to establish traditional contracts for legal viability. Additionally, an intuitive user interface allows for assembling self-aware contracts with building blocks for subsequent parameterization. We recognize that by involving agents, it is possible to process events off-chain and on-chain. That way, we achieve a fine-tuned load balancing where only important events are stored in the blockchain for non-repudiable traceability.

As future work, we aim to develop a mapping from SEC-Language obligations and rights to lower-level so-called smart contract languages such as Soldity that operate directly on blockchain platforms. Furthermore, we investigate a scalable agent-based solution for solving the Oracle problem pertaining to blockchains where a scalable approach assures trusted information is channeled into a self-aware contract collaboration. Important is that the Oracle must be self-healing in that on-the-fly modifications of its constituents are possible in cases of malevolent agent behavior, or contextual changes. Relevant for user adoption is also the design of intuitive graphical user interfaces that allow for laymen such as lawyers, business people, and so on, the development of specific contracts based on human readable templates.

Bio

Alex is an associate professor[5] at TTU.ee and has written the whitepapers for the ICOs of Qtum.org, Agrello.org, Everex.io, CEDEX.com, Datawallet.io, Evareium.io, and more to come.

References

[1] Pleszka K. Araszkiewicz, M., editor. *Logic in the Theory and Practice of Lawmaking*. Springer Publishing Company, Incorporated, 1 edition, 2016.

[2] R.H. Bordini, J.F. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. John Wiley & Sons, 2007.

[3] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml). *World Wide Web Journal*, 2(4):27–66, 1997.

[4] V. Butterin. A next-generation smart contract and decentralized application platform, 2014.

[5] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang. Hermit: An owl 2 reasoner. *Journal of Automated Reasoning*, 53(3):245–269, 2014.

[6] P.A. Hamburger. The development of the nineteenth-century consensus theory of contract. *Law and History Review*, 7(2):241–329, 10 2011.

[7] R. Hull, V.S. Batra, Y.M. Chen, A. Deutsch, F.F.T. Heath III, and V. Vianu. *Towards a Shared Ledger Business Collaboration Language Based on Data-Aware Processes*, pages 18–36. Springer International Publishing, Cham, 2016.

[8] F. Idelberger, G. Governatori, R. Riveret, and G. Sartor. *Evaluation of Logic-Based Smart Contracts for Blockchain Systems*, pages 167–183. Springer International Publishing, Cham, 2016.

[9] Kurt Jensen, Lars Michael, Kristensen Lisa Wells, K. Jensen, and L. M. Kristensen. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. In *International Journal on Software Tools for Technology Transfer*, page 2007, 2007.

[10] L. Luu, D.H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, 2016.

[11] L. Masinter, T. Berners-Lee, and R.T. Fielding. Uniform resource identifier (uri): Generic syntax. 2005.

[12] Business Process Model. Notation (bpmn) version 2.0. *Object Management Group specification*, 2011. `http://www.bpmn.org`.

[13] O. Morten. How firms overcome weak international contract enforcement: repeated interaction, collective punishment and trade finance. *Collective Punishment and Trade Finance (January 22, 2015)*, 2015.

[14] M.A. Musen. The protégé project: A look back and a look forward. *AI matters*, 1(4):4–12, 2015.

[15] A. Norta. *Exploring Dynamic Inter-Organizational Business Process Collaboration*. PhD thesis, Technology University Eindhoven, Department of Information Systems, 2007.

[16] A. Norta and P. Grefen. Discovering Patterns for Inter-Organizational Business Collaboration. *International Journal of Cooperative Information Systems (IJCIS)*, 16:507 – 544, 2007.

[17] A. Norta, L. Ma, Y. Duan, A. Rull, M. Kõlvart, and K. Taveter. eContractual choreography-language properties towards cross-organizational business collaboration. *Journal of Internet Services and Applications*, 6(1):1–23, 2015.

[18] A. Norta, A. Vedeshin, H. Rand, S. Tobies, A. Rull, M. Poola, and T. Rull. Self-aware agent-supported contract management on blockchains for legal accountability. *URL: http://whitepaper. agrello. org/Agrello_Self-Aware_ Whitepaper. pdf*, 2017.

[19] T. Roxenhall and P. Ghauri. Use of the written contract in long-lasting business relationships. *Industrial Marketing Management*, 33(3):261 – 268, 2004.

[20] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.

[21] Nick Russell, Arthur HM Ter Hofstede, David Edmond, and Wil MP van der Aalst. Workflow data patterns: Identification, representation and tool support. In *Conceptual Modeling–ER 2005*, pages 353–368. Springer, 2005.

[22] Nick Russell, Wil MP van der Aalst, Arthur HM ter Hofstede, and David Edmond. Workflow resource patterns: Identification, representation and tool support. In *Advanced Information Systems Engineering*, pages 216–232. Springer, 2005.

[23] I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, and J. Mendling. *Untrusted Business Process Monitoring and Execution Using Blockchain*, pages 329–347. Springer International Publishing, Cham, 2016.

[5] https://tinyurl.com/Alex-Norta