Self-Aware Agent-Supported Contract Management on Blockchains for Legal Accountability

Alex Norta[1,2], Anton Vedeshin[1], Hando Rand[1], Simon Tobies[2], Addi Rull[1,2], Margus Poola[1,2], Teddi Rull[1]

[1] Agrello OÜ
Pärnu mnt. 548-15, 10916 Tallinn, Estonia
hando@agrello.org,addi@agrello.org
[2] Large-Scale Systems Group, Tallinn University of Technology,
Akadeemia tee 15A, 12816 Tallinn, Estonia
alex.norta.phd@ieee.org

Abstract. This whitepaper addresses existing problems with conventional non-machine readable contracts. Such conventional contracts (CC) are complicated to set up, disconnected from ICT-systems and when conflicts occur, tracking their execution is restrictively slow and in addition, CCs are 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. 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. However, currently existing machine-readable contract solutions, i.e., smart contracts, lack suitable obligation constructs for execution and enforcement. Additionally, current systems do not comprehend the dynamics of legal relationships. It is important to mask legal obligations with daily human conduct. This whitepaper address the gap by specifying a so-called Agrello-framework that enables blockchain-driven self-aware agents-assisted contracts for a decentralized peer-to-peer (P2P) economy.

Key words: self aware, multi agent, blockchain, smart contract, decentralized, per-to-peer, e-governance, human readable

CRYP2NOVA

1 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 as evidence, is that the parties involved voluntarily engage to establish a consensus [19]. In most business cases, CCs are documents [44] 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

Archived copy · cryp2nova.com

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 conflict. The resulting costly conflict resolutions may even collapse an entire contractual relationship. Also the enforcement of CCs [29] proves to be either too complicated, time consuming, or impossible, certainly in international circumstances.

The authors in [20] recognize that 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 [25] of Ethereum. In [52], 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 [21], different smart-contract language choices are compared. While procedural languages are currently the norm[25], 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 state of the art above also does not recognize that such SACs must cater for having humans in the contract loop. This paper fills the gap by posing the question how to make self-aware human-readable contracts 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 enables SACs to be human manageable? What ensures contract immutability for legal viability?

The remainder of this whitepaper is structured as follows. Section 2 presents a running case for SAC management along with related literature that prepares for subsequent sections. Section 3 focuses on the important relationship in SACs between essential content and the mapping to business processes that require monitoring. Section 4 discusses the meaningful integration of humans in the SAC lifecycle. Section 5 explores which pre-existing blockchain-technology solutions can be combined in a suitable way for achieving a trustable management of contract elements. Section 6 evaluates the results against the running case, employing a proof-of-concept prototype. Finally, Section 7 concludes the whitepaper and also comprises plans for future work.

Archived copy · cryp2nova.com

2 Background Literature and Running Case

In section 2.1 we present related literature that prepares the reader for subsequent sections. Section 2.2 contains a running contract case that stems from real-life apartment-renting contracts. Note that we use the terms beneficiary for creditor and obligor for debtor.

2.1 Related Work

Scholarly literature about SACs exists. In [2], the core elements of legislation are 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 whitepaper is on obligations the properties of which Figure 1 informally depicts.

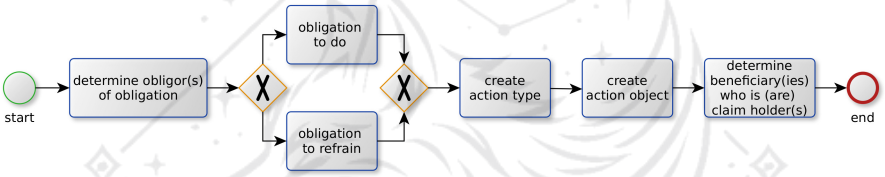


Fig. 1. Informal properties of an obligation.

The properties in Figure 1 show a micro-process for obligations development using the business-process modeling notation BPMN [28]. The small 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. Further details about these so-called smart obligations are presented in Section 3.

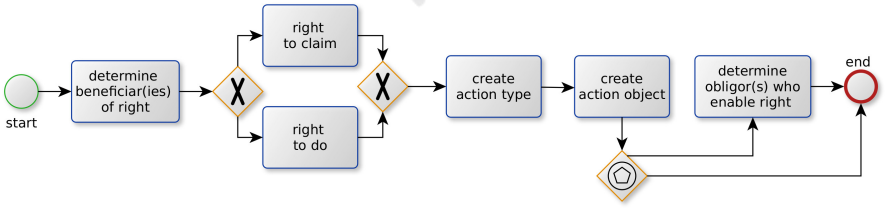


Fig. 2. Right-development micro-lifecycle.

Archived copy · cryp2nova.com

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. 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.

Orchestration and choreography protocols that facilitate, verify and enact agreements between consenting parties are termed smart contracts [23, 40, 45]. The latter initially find application in diverse domains such as financial technology [10], Internet-of-Things (IoT) applications [42] and digital-signing solutions [14]. An essential aspect of smart contracts is a decentralized validation of transactions, initially by means of so-called proof-of-work (PoW) [51]. The core technology that enables smart contracts is a public distributed ledger termed the blockchain that records transaction events without requiring a trusted central authority. Blockchain technology spreads in popularity with the inception of Bitcoin [31], a peer-to-peer (P2P) cryptocurrency and payment system that comprises a limited set of operations on the protocol layer. Bitcoins use PoW for transaction validation that is computationally expensive and electricity intensive.

Most proof-of-stake (PoS) blockchains can source their heritage back to PeerCoin[1] that is based on an earlier version of Bitcoin Core. There are different PoW algorithms such as Scrypt[2], X11[3], Groestl[4], Equihash [6], etc. The purpose of launching a new algorithm is to prevent the accumulation of computing power by one entity and ensure that Application Specific Integrated Circuits (ASIC) can not be introduced into the economy.

There are ongoing discussions about consensus and which platform meets the needs of respective project requirements. The consensus topics most widely discussed are: PoW [50], PoS [5], Dynamic PoS[5], and Byzantine Fault Tolerance [12] as discussed by HyperLedger. The nature of consensus is about achieving data consistency with distributed algorithms. Available options are, e.g., the Fischer Lynch and Paterson theorem [7] that states consensus cannot be reached without 100% agreement amongst nodes.

In the UTXO model, transactions use as input unspent Bitcoins that are destroyed and as transaction outputs, new UTXOs are created. Unspent transaction outputs are created as change and returned to the spender [1]. In this way, a certain volume of Bitcoins is transferred among different private key owners

[1] https://peercoin.net/
[2] https://litecoin.info/Scrypt
[3] http://cryptorials.io/glossary/x11/
[4] http://www.groestlcoin.org/about-groestlcoin/
[5] http://tinyurl.com/zxgayfr

Archived copy · cryp2nova.com

and new UTXOs are spent and created in the transaction chain. The UTXO of a Bitcoin transaction is unlocked by the private key that is used to sign a modified version of a transaction. In the Bitcoin network, miners generate Bitcoins with a process called a coinbase transaction, which does not contain any inputs. Bitcoin uses a scripting language for transactions with a limited set of operations[6]. In the Bitcoin network, the scripting system processes data by stacks (Main Stack and Alt Stack), which is an abstract data type following the LIFO principle of Last-In, First-Out.

In [39], the authors define an ontology that allows for a rapid validation of the concepts and properties existing contracting languages comprise[7]. A state-of-the-art formalization means of ontologies is to use the web ontology language OWL [27]. The latter organizes class hierarchies and allows practitioners to find a common semantical understanding about a problem domain. Note that ontologies represent constantly evolving information on the Internet originating from heterogeneous data sources.

The obligation ontology for this paper we design with the Protégé tool [30] 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 [18] to check the ontology consistency, identify subsumption relationships between classes, and so on.

Since the obligation ontology is static, we employ Coloured Petri Nets (CPN) [22] as a graphical oriented language for covering the dynamic aspects of obligation processing using CPNTools[8]. 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.

The holistic lifecycle management of SACs is relevant and has been ignored so far by industry practitioners. Consequently, in [34], the startup phase commences with choosing from a library a contract template where the latter is configured with service types and roles. Concrete service offers from tentative eCommunity partners populate the service offers and roles before a negotiation phase either results in a terminating dissent of only one party, or a counteroffer that requires a restart of the negotiation, or a consent that establishes a contract. The next phase of the lifecycle [35] involves creating local contract copies for each eCommunity partner. The local contract copies are the means for deducing respective sets of business policies, network monitors, monitoring agents and communication endpoints of concrete technical services for the enactment phase. The latter [41] is carried out in a distributed way and when a violation

[6] https://en.bitcoin.it/wiki/Script
[7] https://steemit.com/smart/@alexbafana/smart-contract-languages-comparison
[8] http://cpntools.org/

Archived copy · cryp2nova.com

of a business policy occurs, the non-violating eCommunity parties must vote on the perceived severity. The outcome options are either calming for the ongoing contract enactment, or disruptive. The former maintain the enactment and comprises voting outcomes, such as ignoring a violation, or replacements of a business rules, service offers, an eCommunity party with a new one, and so on. Calming reactions require to varying degrees a destruction of flowing business semantics for rolling back the remaining subset to earlier contract-lifecycle stages in a targeted way. A disruptive voting outcome leads to a sudden termination of an ongoing enactment as the business-rule violation is perceived as too severe. The rollback results in a new negotiation for starting another contract, unless the business case seizes to exist.

The potential conflicts that occur between decentralized autonomous agents (DAO), require specific modeling, management and resolution [32]. The management works as such that first, a conflict is detected by analyzing the exceptions reported during execution. Second,the conflict type, origin and impact of an exception must be uncovered. Finally, depending on the nature of an exception, the appropriate conflict negotiation and resolution strategy among the participating entities is implemented. An ontology enables modeling conflict types along with related exceptions, negotiation and resolution strategies, thereby enabling conflict management and resolution.

Ongoing contract enactments may also evolve [16] in an orderly way. In that case, the assumption is that a collaborating party modifies an internal technical process that matches with an externally exposed service offer. The latter is a subset of the internal process so that business secrets remain private. Based on a set of rules, such internal process changes may trigger varying changes of a process view that may cross over into the domain of collaborating parties to the degree of affecting other eCommunity-party internal processes. The objective is to assure that a collaboration configuration remains sound in that the enactment of a contract reaches the desired terminal state.

Finally, there is a clear need to bring different solutions from different technology and application domains together for a holistic design of cyberpysical systems (CPS) [43]. CPS integrates computational and physical capabilities that allow for interaction with humans through diverse means [3]. Such novel interaction ways expand the capabilities of humans in correlation with the physical world through computation, communication, and control as a key attraction feature of CPS. For example, in domains in the design and development of next-generation avionics and vehicles, smart cities, Industry 4.0, and so on. The flexible and scalable governance of CPS raises the need for employing SACs where smart contracts are combined with smart objects such as Belief-Desire-Intention (BDI) agents [8].

Jason[9] 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.

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

Archived copy · cryp2nova.com

2.2 Running case

A user story depicted in Figure 3 describes a process of making a rental agreement based on activities of a lessor and a lessee. Lessor is a person who is a property owner or a person who represents a property owner and has a right to make a rental agreement on behalf of an owner. We call him John. Property in this use case scenario is what in legal terms is called an immovable, i.e., a plot of land and anything permanently attached to the plot of land, such as a house, an apartment, a condo or some other type of premises such as a garage, a parking lot, a shed, i.e., it is a space that can be rented out. Property can also be movable in legal terms such as, e.g., a trailer, or a tool. Lessee is a person who is looking for property to rent for a long or a short period of time. We call her Mary.

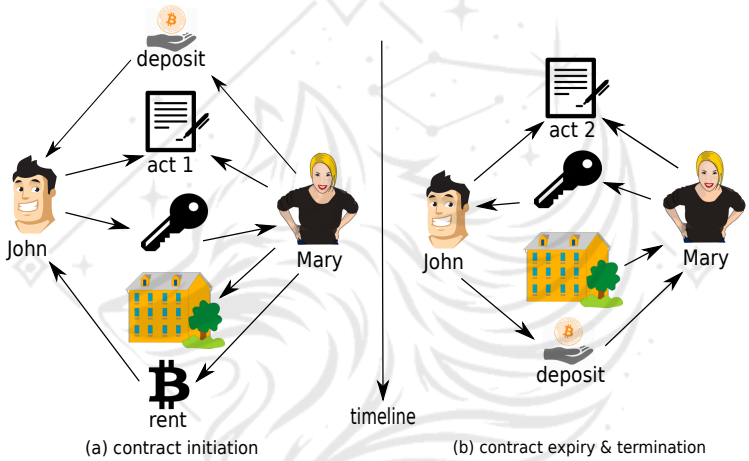


Fig. 3. The informal exchange protocol for (a) the contract initiation and (b) for contract expiry and -termination. For (a), the lessee pays a deposit to the lessor and next both sign a rent contract (act 1). The apartment keys are handed over to the lessee for moving into the apartment, for which the lessee pays monthly rent. For (b), a possession-retrieval act (act 2) is signed that may state the apartment is in the same condition as for (a), the apartment keys are handed back to the lessor, the lessee moves out and finally, the lessor pays back the deposit.

The lifecycle of a rental contract is divided into the following stages/phases: a) preparatory, b) negotiations, c) contract execution e) rollback and e) a contract expiry stage. Following Figure 3, the preparation phase of a rental contract is triggered upon a request from Mary who is looking, for example, for a suitable apartment for the period of 12 months. At this stage, certain standard requirements of the contract have to be determined before John and Mary can proceed to negotiate terms of the contract. Mary needs information about the owner of the property and the apartment as John needs information about Mary. John's and Mary's names, personal identification codes, addresses (data of the parties)

Archived copy · cryp2nova.com

must be included in the contract. The apartment has to be specified so that its condition and status becomes colloquially apparent and formally defined for John and Mary. The object of the contract is defined by characteristics of the apartment such as location (address), size (square meters), intended purpose, e.g., for living, storing, work and for instance whether a parking space or a storage room outside this apartment is included.

Traditionally, Mary looks for information about apartments for rent from the Internet, or she employs a real estate agent to negotiate terms of the contract with John. This means she must spend time to look for information, make calls and visits to find a suitable apartment to rent. A real estate agent would charge a fee for his services.

The Agrello system provides an innovative approach. During the negotiation phase, John has predefined essential terms of the contract (characteristics of the rental object, time, price and rating of the lessee) and based on this information he looks for the best match of a rental request. In the Agrello system, contract conclusion between John and Mary is possible when a match occurs. If Mary declines the offer, she indicates which factors she does consent with. Based on this information John modifies the search criteria for a better match with the modified requirements. The conclusion of the contract means that both parties to the contract have expressed their will to conclude it, i.e., all parties have signed it. This cannot be altered by any third party.

The contract execution phase begins with the documentation of the condition of the apartment. Traditionally, this refers to a possession transfer act that includes information about the condition of the apartment, the recorded state of utilities (water, electricity, heating) and the number of keys given to the lessee. John must hand over the keys and Mary must pay the deposit for the apartment. After that, Mary is obliged to pay the monthly rent on time and to keep the apartment in a good condition. If Mary fails to perform her monthly rent payment then John has the right to claim a late payment charge. A rollback situation occurs if, e.g., John can not provide the apartment any more, then John must return the rent that Mary has payed up front.

The contract terminates if the expiry date of the contract arrives, or if the contract is prematurely terminated. Consequently, the apartment is transferred back to John. During the contract expiry phase, John expects that the condition of the apartment is the same upon return as at the point of initial transferral to Mary. The property-transfer procedure is similar to the description above and the status of utilities must be recorded.

3 Self-Awareness of Contracts

The current smart-contract lingua franca is Solidity. It embodies a programming language that industry practitioners without IT-skills do not comprehend, e.g. lawyers. Therefore, it is not possible to defend such contracts in front of an independent arbitrator due to a lack of suitability, utility and expressiveness in

Archived copy · cryp2nova.com

a legal context. For example, Solidity does not comprise language constructs that resemble obligations and rights pertaining to the parties of a contract.

The purpose of CCs is to establish relationships and to govern the behavior of contracting people, which requires sound constructs of obligations and rights. With the emergence of CPS, smart contracts require the capability of reasoning about rights and obligations, which the involvement of BDI-agents enables. Thus we yield thereby SACs with scalable socio-technical application scenarios where humans use technology for solving problems collaboratively. In CCs, a lawyer has to look at a contract to check if a deadline was missed, or an obligation breached. Self-awareness in this sense means that both entities, the CC and lawyer, merge into one artifact being a software agent that comprises contract logics in the form of machine-readable obligations. More precisely, the agent can deduce, e.g., missed deadlines from the obligations, and since we perceive the agent with the obligations as a smart contract, we conclude a smart contract reasons about itself.

The remainder is structured as follows. Section 3.1 discusses the ontological concepts and properties of contractual obligations. Section 3.2 shows in a formal way the processing of obligations by agents. Next, Section 3.3 explains the use of BDI-agents in managing the contracts.

3.1 Contract Content

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, which assist, automate and manage contract execution. As mentioned above, rights and obligations must be optimized for machine readability. We explain rights and obligations with the running case of Section 2.2. Next, we show machine-readability for rights and obligations while maintaining the capability for non-technical persons to comprehend the smart rights and obligations based SAC.

Figure 4 depicts the class diagram of the Agrello-framework ontology[10]. 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 4 is to eliminate negative consequences that result from a breach of the contract. Additionally, by invoking remedies, a beneficiary achieves a situation if the obligation had been performed

[10] Agrello-OWL: http://tinyurl.com/lkkapvg

Archived copy · cryp2nova.com

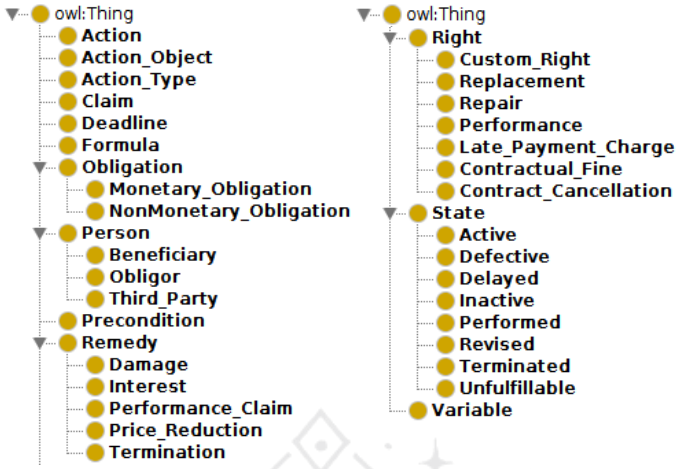


Fig. 4. Agrello-ontology class diagram.

correctly. For example, if the rental payment is delayed, 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 4 reflect the status of an *Obligation* performance in a contract lifecycle.

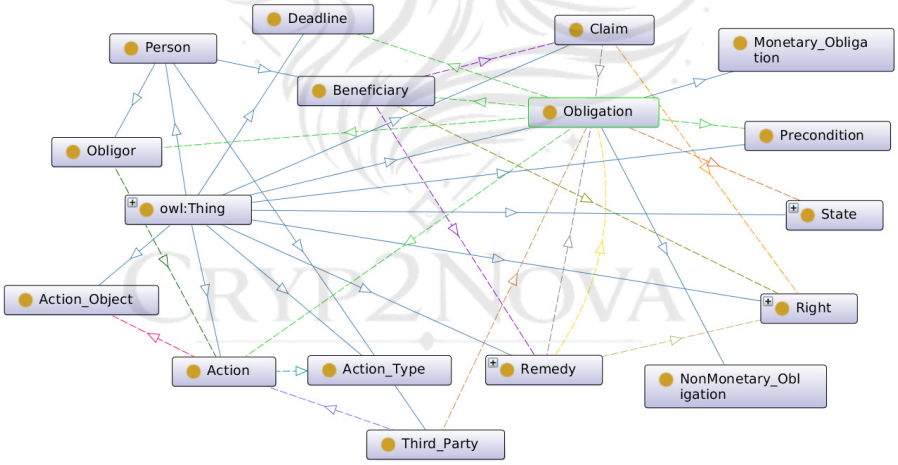


Fig. 5. Obligation-ontology graph.

With respect to class-references, we focus on obligations and rights. In Figure 5 the depicted graph shows the main ontological relationships for the *Obligation*

Archived copy · cryp2nova.com

class. In accordance with Figure 1, the depiction shows components that are integrated with the ontology. More concretely, Figure 5 comprises an *Obligor*, *Beneficiary*, *Precondition*, *Action* and a *Deadline*. Furthermore, a *Remedy* is based on an *Obligation* and a *Third_Party* fulfills an *Obligation*. A *Claim*, or a *Right* may create and *Obligation* and finally, the latter follows lifecycle *State* stages.

In Figure 5, a *Precondition* is an expression that must be fulfilled in order for an *Obligation* to be enabled. An *Action* is the task an obliger must carry out for the *Beneficiary*, e.g. pay the rent. An *Action* has two properties, namely *Action_Type* such as *pay* and the *Action_Object* such as *rent*. The *Deadline* states when an *Obligation* has to be performed.

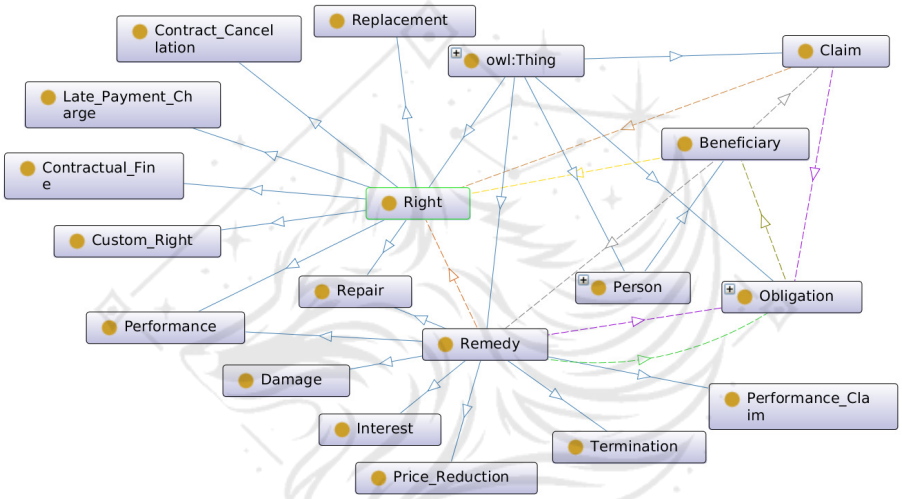


Fig. 6. Right-ontology graph.

The next ontology graph of Figure 6 shows the static relationship of classes related to *Right*. Related to Figure 2, the difference is that a *Beneficiary* has *Right*s prescribed in a contract and a right to several *Remedy* instances if a contract is breached. For example if the apartment is not returned by the lessee in the condition it was in at the beginning of the contract, the lessor has the right to claim repair, or replacement. Damages can always be claimed, irrespective of the aforementioned rights. This can be the case if the lessor can not fulfill an agreement with a subsequent lessee due to delays caused by necessary repairs.

3.2 Obligation Processing

During a contract lifecycle, obligations move through stages of processing. According to the ontology classes of Figure 4, those stages are *inactive*, *active*, *performed*, *delayed*, *defective* and *terminated*. Additionally, there exist the stages

Archived copy · cryp2nova.com

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 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 7, 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 7 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 7 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 7 do not enable the beneficiary to achieve the purpose of the obligation performance, 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.

Archived copy · cryp2nova.com

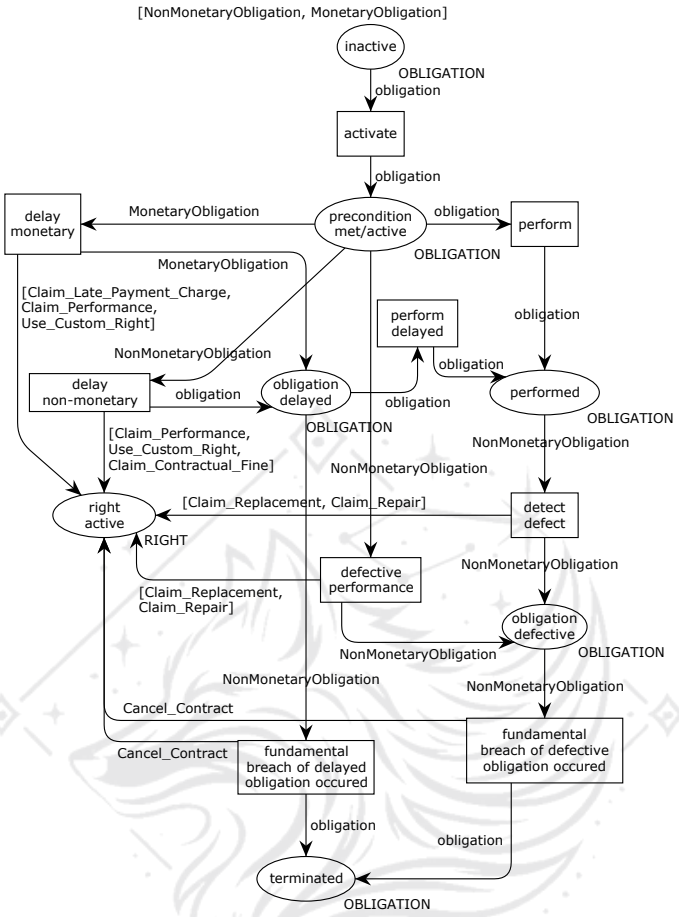


Fig. 7. Transaction processing of obligations.

3.3 Interacting Contract Agents

The running case of Section 2 for renting out apartments is further refined by developing UML sequence diagrams, [46] depicting the interaction protocols of agents. The first sequence diagram of Figure 8 is the refinement for Figure 3(a) about the initiation of a rental contract while the second sequence diagram of Figure 9 pertains to Figure 3(b) about the rental contract termination.

In Figure 8, 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.

Archived copy · cryp2nova.com

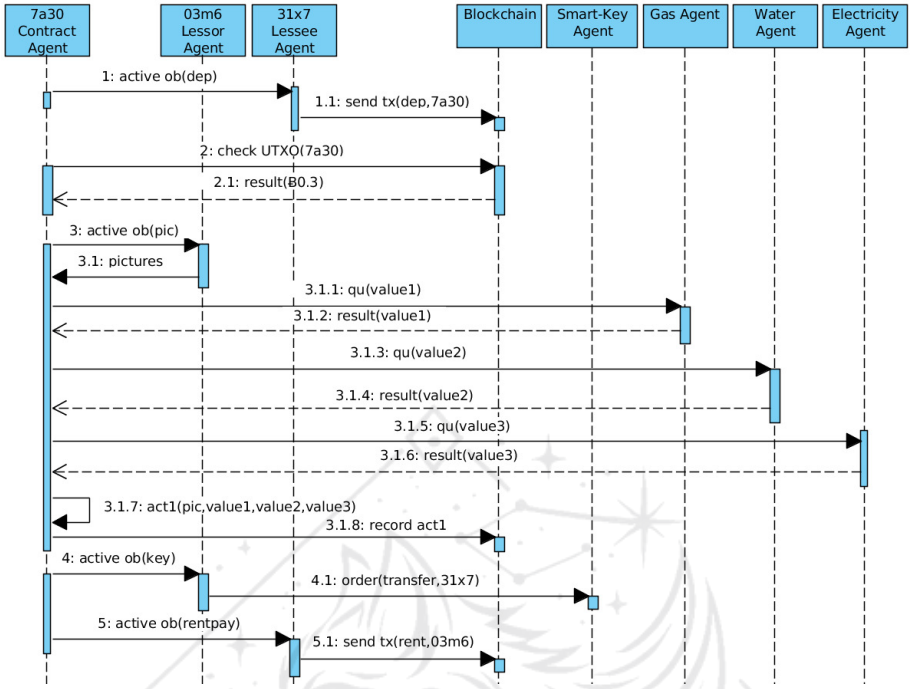


Fig. 8. Initiation protocol of contractual agents.

The sequence diagram in Figure 8 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 8 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(Ƀ0.3)* – indicating the payment to the contract-agent's public-key address on the blockchain.

For the formation of the transfer-act *act1* in Figure 3, 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

Archived copy · cryp2nova.com

counts in *result(value)*-messages. The latter are used by the contract agent in combination with the pictures to generate *act1* 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.

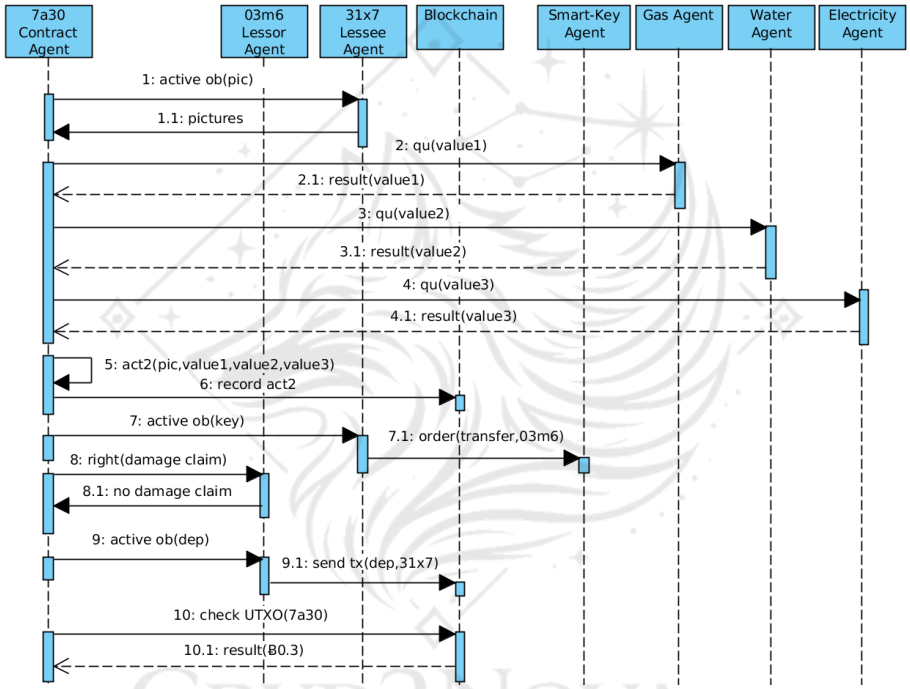


Fig. 9. Termination protocol of contractual agents.

The termination protocol for the apartment rental contract in Figure 9 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 *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 obliga-

Archived copy · cryp2nova.com

tion 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 9 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(Ƀ0.3)*, i.e., the deposit has successfully been returned to the lessee.

4 Manageability of Self-Aware Contracts

The aim of the Agrello-framework is to increase the productivity of information- and value logistics. Important is an understanding of the lifecycle that must be in place for creating, enacting, rolling back and orderly terminating SACs. Consequently, Section 4.1 describes the SAC lifecycle, followed by Section 4.2 that focuses on the involvement of BDI-agents in the lifecycle. Finally, Section 4.3 discusses human-interaction means with the SAC lifecycle.

4.1 Lifecycle of Self-Aware Contracts

For the lifecycle in Figure 10, we use again BPMN notation. The lifecycle commences with the need to establish a peer-to-peer (P2P) contract collaboration between several parties. The first sub-process is for preparing [34] a contract template that is equipped with service types and affiliated agent roles. Thus, we assume a library exists of third-party generated rental contract templates where predefined parameters are inserted, such as for the upper- and the lower bound of rent that an apartment should be offered for. Next, the service types are populated by concrete service-offers from agents that fill specific roles. For the running case of this paper, the roles are lessor and lessee, a blockchain, the smart key, and the utility agents for gas, water and electricity.

Using a SAC approach has advantages over the traditional renting situation as follows. In the latter case, the lessor usually determines who the utility providers are and the lessee is on the receiving end paying to the lessor. In the case of SACs, it is possible to avail specific roles and service types for competing potential providers. For example, if a lessee is conscious about the environment and disagrees with a preset gas agent, there can be provisions to have environmentally friendly alternative providers compete for being gas-service providers. Consequently, fine-tuned free-market competition during the rental-contract preparation phase increases the likelihood of finding an optimal service-provider quorum.

Archived copy · cryp2nova.com

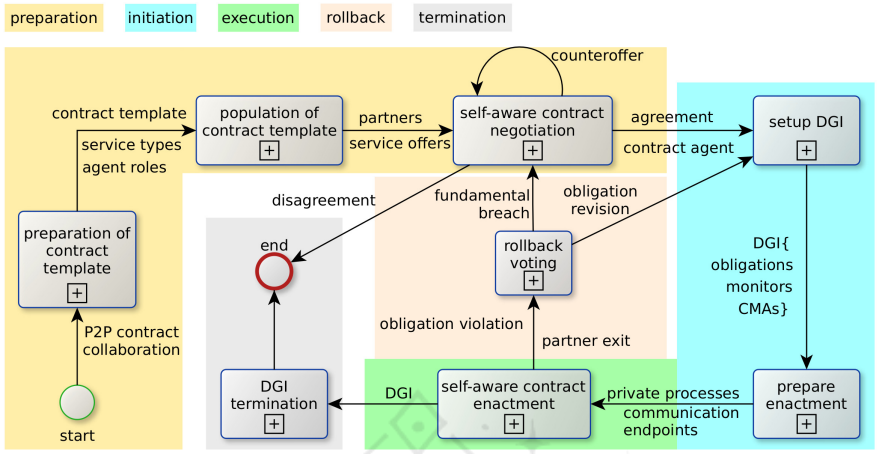


Fig. 10. The Agrello-lifecycle of SACs.

The negotiation sub-process in Figure 10 allows the agents to set concrete prices within the predefined ranges. After assembling a proto-contract, each agent receives a copy for deciding on a negotiation outcome. There are three decision options that may occur during the negotiation. First, the lessee agent may disagree with a set monthly rent rate and proposes a different number in a counteroffer. This implies that new copies must be assigned to each agent for a new negotiation round. Second, an agent considers the proto-contract disagreeable and collapses the negotiation. In this case, the lifeycle reaches the lifecycle end. Third, all agents agree and create a SAC agent.

The initiation phase [35] in Figure 10 commences when the contract agent exists but the establishment of a decentralized governance infrastructure (DGI) is required for the subsequent enactment as well. Note that the initiation phase matches with the sequence-diagram protocol of Figure 8. The DGI-establishment involves the distribution of obligation sets that are derived from the contract agent to the respective agents. Additionally, monitors are assigned together with contract-monitoring agents (CMA) that observe if obligations are adhered to. In cases of obligation breaches, a CMA reports to the contract agent and rollback steps commence that we explain below. Finally, preparing for rental-contract enactment means that private processes for each collaborating agent are set up on a technical level and communication endpoints are established where clarity exists on the meaning of exchanged heterogeneous data sets.

During the enactment phase in Figure 10, the lessee pays monthly apartment rent to the blockchain. The earlier established CMAs monitor if the lessee agent adheres to the deadlines of obligations. In case of an obligation violation [24, 35], a voting procedure commences to establish if the obligation breach is fundamental, or if merely an obligation revision is required. For the running case, a breach occurs when the lessee pays rent late or not at all. Assuming the rent payment

Archived copy · cryp2nova.com

includes also the payment of monthly utility costs, the voting involves the lessor agent and the utility agents for gas, water and electricity. We assume the voting power is determined by the proportional amount of payment that are part of the monthly rent payment of the lessee.

In case the lessee refuses to pay, the voting outcome is a fundamental breach. Consequently, the contract is renegotiated to hopefully clarify issues that result in not paying rent. Note that a renegotiation involves a partial termination of the flowing business semantics and a targeted rollback of the remaining subset into the negotiation component. Still, the existing contract remains intact during the renegotiation phase. If a renegotiation fails, the ongoing rental-contract business-semantics flow is fully terminated and the involved agents are equally terminated.

If the lessee is late with the monthly rental payment to the blockchain, the vote outcome is likely to demand payment on the one hand, and interest in addition on the other hand. We assume revision of the obligation takes place to see if it possibly does not match the rental context any longer, e.g., the lessee receives salary later and consequently, can only pay monthly rent later too. In that case, the obligation adjustment is inserted during a rollback to the initiation phase and the ongoing contract continues. Finally, the full rental-contract termination phase involves a takedown of the DGI and follows the protocol of Figure 9. Thus, all agents are released from the collaboration and equally terminated.

4.2 BDI-Agent Involvement

The agents shown in Figure 10 fulfill specific roles for the self-aware rental contract. The *contract agent* (CA) operates on behalf of the housing agency and coordinates the remaining agents after its creation at the end of the preparation stage when the other agents of Figure 8 consent on a rental-contract establishment. The responsibilities are to allow for deducing a DGI so that the lessor- and the lessee agents comprise local sets of obligations. The CA also coordinates the CMAs that observe locally on behalf of the CA if the local obligation sets are adhered to. Furthermore, the CA has responsibilities during the initiation phase of Figure 10 that follow the sequence diagram of Figure 8. During the execution phase, the CA listens to the CMA and if the rent payment is not performed in an orderly way, the responsibility of the former is to trigger a voting procedure that leads to earlier explained rollback results. Finally, the CA also captures the termination request for the rental contract of the lessor, or lessee and triggers the overall DGI-dismantling. Essential CA-constraints are that the roles of the contract template must all be populated in the preparation stage with corresponding agents. The latter must reach a consensus so that a CA is instantiated for a DGI-setup. Important is also that the data for *act1* creation is delivered as requested, i.e., pictures and gas-, water- and electricity values from the utility agents. Another constraint is that all votes are cast as required during the rollback phase.

The *lessor- and lessee agents* both have the responsibilities to populate a corresponding role with an affiliated service type in a contract template during the

Archived copy · cryp2nova.com

preparation phase. Next, both agents must participate in the negotiation phase and provide counteroffers, disagreements, or agreements for consensus formation.

Both, lessor- and lessee agents, have the responsibility to cooperate for facts collecting that leads to the establishment of *act1* and *act2* in accordance with Figure 8 and Figure 9 respectively during the initiation phase. The lessor agent has the responsibility to transfer the smart key to the lessee. During the execution phase, it is the responsibility of the lessee agent to pay monthly rent onto the blockchain. When the latter obligation is not adhered to, the lessee agent is obliged to cooperate with the rollback procedure and must either agree with a new obligation to continue in the rental contract, or pay damage and compensation if a fundamental breach occurs. The lessor agent has the responsibility to cooperate in the rollback voting procedure if the CA detects a breach that a CMA reports.

The constraint of the *lessee agent* is the ability to pay the deposit during the initiation phase, the rent during the execution phase, possible damages and compensations during the rollback phase. For the termination, the constraint for the lessee agent is that the delivered photos display an apartment condition that is similar to when the lessee moved in and if the condition is lower, the lessee agent must pay compensation to the blockchain address of the lessor agent. The constraint of the lessor agent during the initiation phase is delivering pictures to the CA that document the apartment condition and fails to transfer the smart key to the lessee within an acceptable time limit. During the termination phase, the lessor agent must detect damages in the apartment within an acceptable time limit.

The *utility agents* for gas-, water- and electricity have the responsibility to fill their respective roles in the contract template during the preparation phases and must report current meter values during the initiation phase. In case the lessee agent fails to pay the rent where we assume the utility expenses are a part of it, the utility agents must participate in the rollback voting procedure. Finally, during the termination phase, the utility agents must again deliver meter values for the finalization of *act2* (Figure 9). The constraints are, that utility agents fail to adhere to their responsibilities within given time limits.

The *smart-key agent* has the responsibility to accept being assigned to lesses during the initiation phase and being transferred back to the lessor during the termination phase. Otherwise, the smart-key agent does not have any additional responsibilities during the SAC lifecycle of Figure 10. The only constraint for the smart-key agent is to immediately respond to user-change commands. Finally, the blockchain is not an agent but merely an immutable event record-ledger for the rental contract lifecycle.

4.3 Means of Human Interaction

The organization model of Figure 11 shows human involvement in the collaboration for the running-case rental contract. An organization model is part of the agent-oriented modeling notation [49] and denotes the relationships between human- and BDI-agents. The arcs between the agents specify the relationship

Archived copy · cryp2nova.com

types. In Figure 11 we use *Controls* to show a subordinate relationship between agents, *IsPeerTo* to define equal roles and *IsBenevolentTo* as a relationshp between self-interested agent roles. Additionally in Figure 11, we specify that a utility agent can be either a gas-, water-, or electricity agent.

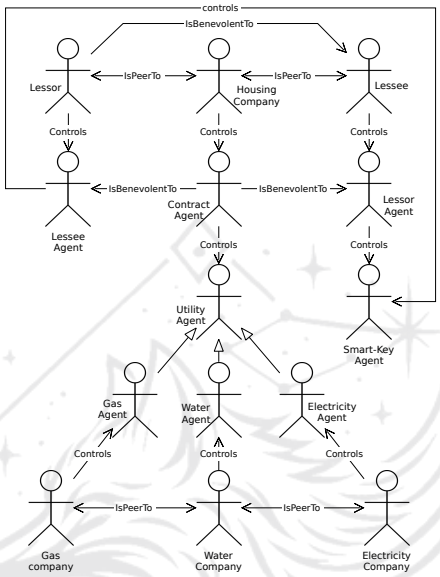


Fig. 11. The organizational model for the rental contract.

The human involvement as specified in Figure 11 takes place via dedicated assisting BDI-agents. For example, the lessee agent engages in the rental contract on behalf of the lessor, who is a human individual. The gas agent acts in the rental contract on behalf of a utility company that provides gas to the apartment. The respective assisting BDI-agents that act for human agents are under coordinating control of the contract agent that are in turn under the control of a housing company. Finally, the smart-key agent is under alternating control of the lessor-, or the lessee agent with respect to shifting the human-agent ownership.

5 Trusting Contract Elements

Contract immutability and legal viability are related to several problems. Immutability not only implies to store the machine-readable and agent-executable contract with its obligations so that it cannot be changed. It is also necessary to store the events, e.g., payments that affect contract-execution immutably. With conventional contracts, these events are for example receipts, emails, or phone calls, e.g., from lessor to lessee. Immutability of these events prevents situations

Archived copy · cryp2nova.com

where two parties claim the opposite about whether a payment is performed, or not. It is also a prerequisite to allow for an agent-aided review of the contract execution. A separate contract agent is fed with the contract in question and connected to the event-storing repository. The agent commences in the past, consumes relevant events and processes the obligations of the contract to the present. Besides immutability of value, this entails that events are immutably timestamped.

The remainder is structured as follow. Section 5.1 comprises essential elements for the setup of SACs. Section 5.2 gives technical details about executing contracts. Section 5.3 describes additional trusted events for contract execution and finally, Section 5.4 explains the contextual trustworthiness for contract integration.

5.1 Elements of Trust

In order to support the execution of self-aware smart-contracts with blockchain technology, various elements are necessary. First of all, the contract has to be signed by the contract parties. Essential elements for enacting blockchain enhanced self-aware smart-contracts are:

Identity: the contract parties must be unambiguously identifiable. Especially the lessor in a rent contract wants assurance that the flat is used by the person who (or who's agent) reacts to the lessor's offer. Assuming that one person with a positive credit rating reacts to a flat offer while the identity used for contract signing cannot be verified, the intended lessee could instead let another person with insufficient credit rating move into the flat, e.g., a friend, or family member.

Signature: conventional contracts require a hand written signature. To achieve comparable, or better legal commitment, digital and cryptographically secure signatures are necessary.

Events: as stated above, access to external events related to the contract obligations is essential for automated execution. In case the contract agent is to sense that a rent happens for a specific month, it either has to be externally informed about such an event with a push message, or must be able to query a blockchain that stores such events as a pull message. Alternatively, an agent with blockchain connection to the blockchain relays the information transitively. The different types of events are not necessarily stored on the same blockchain as, e.g., sensor data from smart meters, or access permissions for the smart lock may require a different blockchain.

Timestamp: the SAC together with agents and obligations considers external events to process obligations, e.g., payments. To reason about delays and deadlines, events consumed by the contract agent need to be time stamped.

Contract source code: the source code of the contract contains its obligation in a formal and machine-readable notation. To guarantee immutability, the contract source code and the corresponding hash must reside on the blockchain. In case a contract party claims defective contract execution by the agent

Archived copy · cryp2nova.com

because of manipulated source code, an original copy must exist to resolve such conflicts. Storing the hash on the blockchain is only sufficient if at least one contract party provides source code with the same hash value. An edge case occurs when both parties can not provide a copy of the contract.

5.2 Contract Execution

Regarding legal viability, it has to be considered where the execution of the contract takes place. Examples for related issues are, the number of independent nodes running the contract and the problem of on- versus off-chain execution.

Unlike Etherum smart contracts, SACs do not require external events, or the progress of obligation executions to be stored on a blockchain. It is legally sufficient to merely store transactions between contract parties on a blockchain. Still, the current state of SAC execution can always be derived from blockchain-stored information.

Processing the rent-payment obligation of Listing 1 requires handling logics of deadlines. Thus, the obligation (*l.1-4*) states that the lessee (obligor) has to pay monthly rent to the lessor (beneficiary). The time token (*l.6*) is stored in the agents belief base to trigger the processing of the recurring `o6` obligation. In an agent-orient approach, the respective code is provided as a library, or module for which Listing 2 provides an excerpt. In *l.1-9:* the plan handles the instantiation of the rent payment obligation each first day of a month. In *l.12-17:*, to acquire the date for the following month, this plan is used. The plan in *l.20-30:* is invoked by lines 1-9 with the purpose to create an event inside the agent that triggers the creation of a concrete obligation at the beginning of the following month. Thus, the event triggers plan l.1-9 again. If the event occurs, a term such as in Listing 1, l.6 is added to the belief base.

Listing 1. A monthly rent-payment obligation.

```
1 obligation(o6, lessee, lessor,
2 date_precondition(year,month,1), pay(rent)
3 )[recurring_deadline(date(year,month,10)),
4 state(new),recurring].
5
6 time_token_for_recurring_obligation(o6,timeToken(2017, 2, 1)).
```

The logic of this micro lifecycle described in Listing 1 is based on the declarative and logic programming facilities the Jason agent framework[11] comprises. Integrating the agent with the blockchain leads to a trade-off raising the question of whether to run the contract agent on-, or off-chain.

Off-chain: running the SAC off-chain requires storage on a server for the execution by the agent framework. This entails, that the agent framework provides means for communicating with the blockchain. The Jason framework with its

[11] More precisely: AgentSpeak is a logic- and declarative programming language. Jason agents are written in AgentSpeak and running them means that the source code is interpreted by the Jason framework.

Archived copy · cryp2nova.com

reasoning cycle is implemented in Java and allows for extending it with hand-written code, i.e., using a library for interfacing with the blockchain. For this to work, the blockchain has to provide an up-to-date API that also introduces a dependency on it for the agent, not to speak of Java being involved. The off-chain approach does not require any adaption, or change of the used blockchain, but only the API.

Listing 2. Instantiation of the rent payment obligation each 1st day of a month.

```
+time_token_for_recurring_obligation(ObligationName, timeToken(OY,OM,OD))
    <-
    +obligation(ObligationName, Obligor, Beneficiary,
        date_precondition(OY,OM,OD), Task )[recurring, state(new)];
    ?obligation(ObligationName, Obligor, Beneficiary,
        date_precondition(Y,M,D), Task )[recurring]
    ?getNextTimeToken(timeToken(OY, OM, OD),
        date_precondition(Y,M,D), NewDate);
    !create_event_for_recurring_obligation(ObligationName,NewDate).


+?getNextTimeToken(
    timeToken(TY, TM, TD),
    date_precondition(year,month,Day),
    NewDate) : .number(Day) & TM <= 11
    <-
    NewDate = date(TY, TM +1, Day).


+!create_event_for_recurring_obligation(ObligationName,
    date(NY, NM, ND)) : sulfur.date(A,B,C) & a(NY,NM,ND) <= a(A,B,C)
    <-
    .concat(
        "+time_token_for_recurring_obligation(",
        ObligationName,
        ",",
        timeToken(NY, NM, ND),
        ")",
        Event);
    .at("now +1 ms", Event).
```

On-chain: comparable to Ethereum smart contracts. For obligation driven contracts, it is not necessary to write any state information to the blockchain. The contract execution is driven by events such as payments that need to be stored on the blockchain. Executing the contract requires keeping the execution state in the memory of the machine/hardware for execution. The higher degree of abstraction that a logic- and declarative programming language such as AgentSpeak yields, introduces the necessity for a reasoning cycle, e.g., reacting to percepts, and a reasoner for unifying terms. To run a SAC on-chain, these components have to be implemented in the virtual machine of the blockchain that requires developing a new language [12]. This eliminates the need for API integration into an agent. Another advantage is the encapsulation and hiding of the hand-written code that comprises the mirco-lifecycle for general obligation processing as in excerpt in Listing 2.

[12] Another way to tackle this problem is introducing an intermediate language based on π-calculus.

Archived copy · cryp2nova.com

An open issue is the relation between declarative and imperative programming in smart contracts. The SAC language we aim for is based on obligations and a more static and declarative approach. This leaves open the question whether logics like voting protocols/algorithms can also be covert this way.

The question arises which parties run the contract agent. On the one hand, contract details may not be accessible to the public while on the other hand, the idea of blockchain is to distribute information for tampering protection and traceability. One solution is that each contract party runs a light node for contract-code enactment that only stores those transactions related to the contract and its parties.

5.3 Trusted Events

When a SAC is executed, not only events like payments are involved. The agent, reasoning about obligations, also needs to keep track of fine grained information bits. If a recurring obligation must be processed such as for a monthly rental payment, the agent must create a concrete obligation[13] each first of a month, containing the according date (Listing 2 line 3).

The agent senses its environment checking the current date and recognizes that the recurring obligation has to be instantiated since the date is the first of the month. This creates additional information the contract agent requires to appropriately process the obligations. In contrast to payment events and given signatures, or value transfers, this information does not require storage in the blockchain and can be restored by restarting the contract agent, e.g., starting a second instance to review contract execution.

5.4 Contextual Trustworthiness

Self-aware contracts can be integrated with their context to different degrees. For a rental contract, an especially high degree of integration is computing the monthly utilities by reading the consumption of electricity and water from smart meters. This raises difficulties on several levels.

Security: IoT devices have repeatedly been subject to serious security vulnerabilities[14]. Paying utilities automatically may cause inconveniences for a lessee

[13] We clarify the terminology of recurring and concrete obligations. In a rental contract, there is only one obligation for paying the rent monthly. However, for processing, this obligation must be split into parts (instances) simplifying the task for the agent.

[14] `https://techcrunch.com/2017/02/28/amazon-aws-s3-outage-is-breaking-things-for-a-lot-of-websites-and-apps/`
`https://security.radware.com/ddos-threats-attacks/brickerbot-pdos-permanent-denial-of-service/`
`https://krebsonsecurity.com/2016/10/who-makes-the-iot-things-under-attack/`
`http://iotworm.eyalro.net/`

Archived copy · cryp2nova.com

of a flat. The logics for recognizing and handling faulty sensor data must be introduced to the contract agent. A privacy concern is the secure transfer of sensitive measurement data.

Interoperability: when integrating technologies from different technical domains[15] such as software agents, blockchains, smart devices/IoT, a considerable challenge is data exchange. This not only relates to syntactic interoperability so that exchanged data can be parsed by the counter-party with correct data formats. The semantic interoperability for data exchange must also be clarified, e.g., a temperature measured in °C is mistakenly interpreted as °F ?). Sound data exchange also involves *architectural* aspects. The question arises if the contract agent receives the sensor data from another software agent that acts as a proxy for the IoT devices and provides a uniform interface, or is accessing the smart meters hard coded into the agent.

Our notion of SACs is based on obligations and we refer to the P2P economy throughout the white paper. Nevertheless, the idea of obligations stems from contract law that not only covers P2P scenarios such as for the rental-contract case. Obligations are also applicable for business-to-business scenarios. We anticipate a framework that is scalable for business-to-business (B2B) cases without considerable changes to the core concepts, e.g., sale of goods, or services.

6 Feasibility Evaluation

We map the running rent-contracting case of Section 2 into an evaluation that comprises three parts. First, Section 6.1 gives an architecture of the Agrello framework. Next, Section 6.2 shows code of the Agrello language for several aspects of the SAC, which includes examples for obligations and rights. Section 6.3 shows a conceptual Agrello graphical user-interface prototype that is currently under development.

6.1 System Architecture

The Agrello-system architecture in Figure 12 we derive from from [37] where the so-called eSourcing Reference Architecture (eSRA) is introduced for cross-organizational process-aware collaboration. We use a simplified form of a UML-component diagram model [4] for the architecture depiction. Briefly, eSourcing establish outsourcing relationships by supporting matchmaking between offered and requested process views [17] that allows for a stepwise collaboration evolution if contextual changes require it. Furthermore, in [32] we show how software agents enable communication for conflict resolution in P2P- and process-aware collaborations. These agents act autonomously on behalf of collaborating organizations a SAC orchestrates with the assurance that information used for

[15] `http://www.jot.fm/issues/issue_2006_11/article4/`

Archived copy · cryp2nova.com

conflict resolution is trustworthy. Note that business processes can be mapped onto Solidity [52] for blockchain-based enactment.

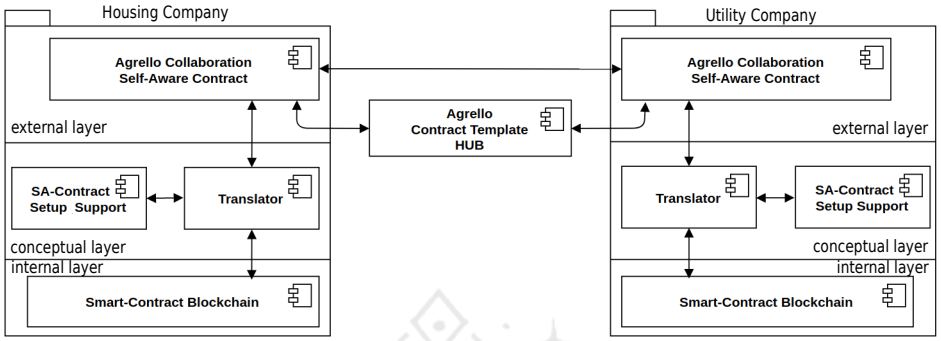


Fig. 12. Agrello-system architecture in a simplified bi-lateral cross-organizational collaboration scenario.

The architecture in Figure 12 shows a set of components that are organized with a layered architecture. We assume to the left and right are two respective collaborating parties. However, the eSourcing approach scales so that Agrello supports the P2P multi-party running case [15, 16] from Section 2. In the middle of Figure 12 is a component that comprises Agrello contract templates in a repository hub [38]. The hub allows for finding contract templates that a GUI displays in human-readable language during the preparation of a contract template in correspondence to Figure 10. A user parameterizes the template and a mapping creates a representation into an XML-based based equivalent based on the eSourcing Markup Language (eSML) [39] that we extend with obligations and rights into *AgrelloLanguage* in accordance with Section 3. Note that for the population and negotiation phases in accordance with Figure 10, the collaboration components are employed.

The layers in Figure 12 are as follows. The external layer comprises a respectively replicated collaboration component that synchronizes via a contained coordination interface with the equal component of counter-parties. The collaboration component also serves as a security-ensuring gateway for P2P data exchange. Furthermore, embedded are additional components to negotiate SACs specified in AgrelloLanguage with affiliated obligations and rights. For that, functionalities are necessary to perform reputation-, identity-, and action management related to SAC involvement. The collaboration component also contains BDI-agents that act on behalf of a concrete collaborating party, as we explain above. A coordinator component assures during the enactment phase that SACs align the execution on obligations, rights and the the process aspect.

The contract-template hub comprises several embedded components too. Identity-management- and reputation management functionalities complement the collaboration components of respective collaborating counter-parties. Bid-

Archived copy · cryp2nova.com

ding services receive auctioning- and tendering input from the collaboration management. To quickly deploy contracts, libraries for SAC templates, obligations and rights exist. Important is also that the templates must be validated with tool support for soundness, i.e., syntactic and semantic deficiencies elimination.

The conceptual layer is internal to collaborating parties and shielded by the external layer from information exchange with counter-parties. The contained SAC setup support comprises libraries with smart-contract parts, obligations and rights that modeling tools can use for rapidly creating SACs. The composed SAC is simulated and verified internally for soundness, i.e., for eliminating design errors. Note that simulation is inferior to verification as most likely not all execution paths of a SAC are tested. In contrast, verification employs formal methods and sophisticated tool support for a complete correctness check of SACs. The setup support also comprises development tools for issuing BDI-agents that externally represent a collaborating party. The translator component relays between the external- and internal layer that we explain below.

As [37] shows in detail, the translator must reconcile different standards and formats that are used in terms of modeling notations and information types. While the external- and conceptual layers are provided by the Agrello framework, the internal layer comprises concrete smart contract systems, e.g., Ethereum [53], Lisk[16], Hyperledger [12], Qtum[17], and so on. Thus, we assume a blockchain agnostic position and must map to a heterogeneous set of smart-contract languages, mostly slight variations of Solidity [13].

6.2 Agrello Language

We show the high-level structure of the business-collaboration language we call AgrelloLanguage that is derived from the research-driven and pre-existing eSourcing Markup Language (eSML) [39] schema as a foundation. AgrelloLanguage is given in Extensible Markup Language (XML) [9] to facilitate the building of distributed applications in Clouds [11].

Figure 13 shows the structure of AgrelloLanguage as a SAC between collaborating parties, structuring the AgrelloLanguage 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) [26].

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.

[16] https://lisk.io/
[17] https://qtum.org/en/

Archived copy · cryp2nova.com

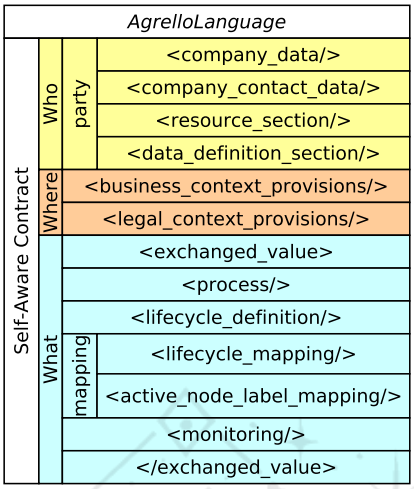


Fig. 13. AgrelloLanguage structure for SAC formulation.

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-flow patterns for business-process definitions that have semantic clarity [36]. Note that the process definitions comprises constructs for linking to the resource- and data-definition sections of AgrelloLanguage that are both based on respective pattern collections [48, 47]. Furthermore, life-cycle definitions [33] are for the business processes and contained tasks.

Since the Agrello 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 13 specifies how much of the enactment phase the service consumer perceives. We refer the reader to [33] for further details.

For the running rental case, we give brief AgrelloLanguage code examples for obligations and rights. Listing 3 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 3 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

Archived copy · cryp2nova.com

number of the lessor wallet from Figure 8. 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 3. 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 3, 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 4 comprises intersecting specification elements with an obligation. As pointed out in Section 2, the main difference with an obligation is the the beneficiary may waive a right. We assume in the right example of Listing 4 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 3.

Archived copy · cryp2nova.com

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 4. 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 4 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.

6.3 Agrello User Interface

The objective of the Agrello-framework is to allow for an intuitive development of SACs that are mapped to technical lower-level representations in AgrelloLanguage that is further mapped onto, e.g., Solidity to operate directly on a blockchain. In the conceptual interface depiction of Figure 14, a template builder shows to the left SAC-blocks to drag and drop into a contract window at the bottom right. At the top right of Figure 14, a window for parameterizing the blocks shows, e.g., the *amount* parameter is set to $1.750 as monthly rent.

The bottom left of Figure 14 depicts functions that represent actions in a contract a respective party must carry out. For example, the variables and functions are used in the contract template at the bottom right comprising the function

Archived copy · cryp2nova.com

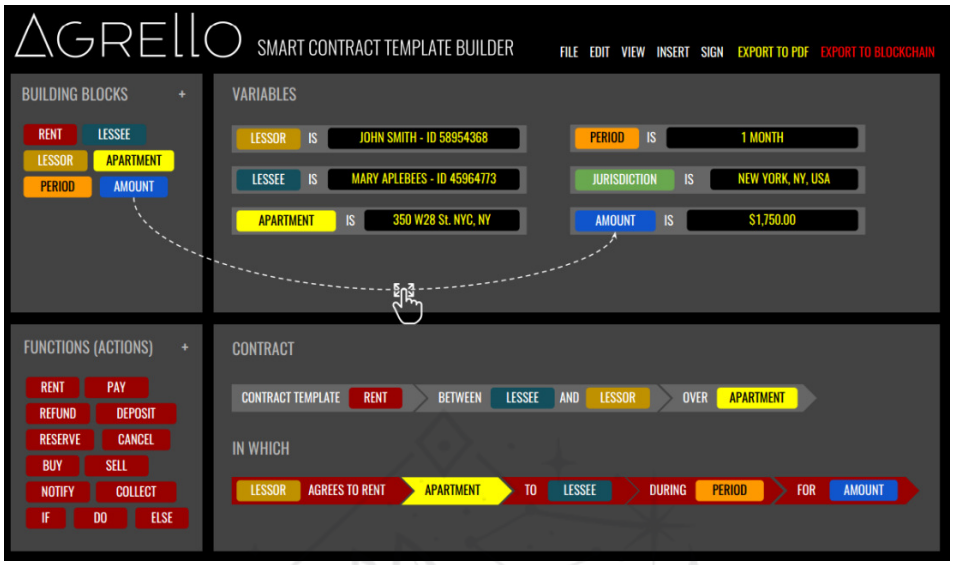


Fig. 14. An intuitive Agrello graphical user interface for SAC-development.

rent involving specified variables *lessor* and *lessee* over a variable *apartment*. More concretely, the contract window shows an assembled obligation in which the defined *lessor* agrees to rent the *apartment* to the *lessee* during the specified *period* for the *amount* of $1.750 per month.

7 Conclusions

This whitepaper presents a novel cross-organizational blockchain-agnostic framework for peer-to-peer collaboration that is based on ca. 15 years of academic research stemming from the first author. With the emergence of cyber-physical systems, a potential arises to reduce costs and time spent on information- and value-transfer logistics that so far humans have managed. Novel blockchain technology enabled smart contracts, combined with intelligent multi-agent systems and internet-of-things devices, yield so-called self-aware contracts that allow for a high degree of automation for such peer-to-peer collaborations. We demonstrate the approach in a running case for renting an apartment that is first presented with traditional protocols for initiating and terminating a rental contract. Since existing blockchain-based solutions lack essential constructs for specifying legally binding, machine-readable contracts, we pragmatically formalize obligations and rights with an ontology. The running case is next mapped onto an automated protocol where belief-desire-intention agents act on behalf of humans who can consequently focus on decision making via mobile devices. 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

Archived copy · cryp2nova.com

traceability. Next, AgentSpeak code-samples indicate the way how belief-desire-intention agents act on behalf of humans to facilitate information- and value-transfer logistics. Important is that the AgrelloLanguage 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 using, e.g., Solidity.

We discover that the combination of belief-desire-intention agents together with the declarative AgrelloLanguage yields self-aware contracts where the former assure as a combined set trusted information is channelled into contract-based collaborations. That way, the agents create a composed oracle governed by a lifecycle-management layer. The latter comprises the stages for preparing a self-aware contract template, initiating the setup phase of a collaboration, enacting a contract, managing rollbacks that are caused by e.g., a breach of an obligation, or the deliverance of faulty information by an agent, and an orderly termination of a self-aware contract collaboration.

In addition to employing agents that provide a degree of artificial intelligence in a collaboration, human manageability of the Agrello framework we achieve by providing a declarative smart-contract language that specifies cross-organizational contract-collaborations. This AgrelloLanguage is based on a pre-existing language that results from an EU-project for initially automating cross-organizational production processes. The AgrelloLanguage 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.

Immutability for legal viability the Agrello framework achieves by employing blockchain capability. Contracting parties must be signed digitally after the parties' identities are authenticated. Furthermore, relevant external events are stored on the blockchain together with their respective timestamps that are critical for assuring legal traceability. Also the actual contract code itself we store on the blockchain to guarantee immutability. 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 AgrelloLanguage 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

Archived copy · cryp2nova.com

laymen such as lawyers, business people, and so on, the development of specific contracts based on human readable templates.

References

1. A.M Antonopoulos. Mastering bitcoins, 2014.
2. Pleszka K. Araszkiewicz, M., editor. *Logic in the Theory and Practice of Lawmaking*. Springer Publishing Company, Incorporated, 1 edition, 2016.
3. R. Baheti and H. Gill. Cyber-physical systems. *The impact of control technology*, 12:161–166, 2011.
4. D. Bell. Uml basics: The component diagram. *IBM Global Services*, 2004.
5. I. Bentov, A. Gabizon, and A. Mizrahi. *Cryptocurrencies Without Proof of Work*, pages 142–157. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.
6. A. Biryukov and D. Khovratovich. Equihash: Asymmetric proof-of-work based on the generalized birthday problem. *Proceedings of NDSSâĂŹ16, 21–24 February 2016, San Diego, CA, USA. ISBN 1-891562-41-X*, 2016.
7. B. Bisping, P.D. Brodmann, T. Jungnickel, C. Rickmann, H. Seidler, A. Stüber, A. Wilhelm-Weidner, K. Peters, and U. Nestmann. Mechanical verification of a constructive proof for flp. In *International Conference on Interactive Theorem Proving*, pages 107–122. Springer, 2016.
8. R.H. Bordini, J.F. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. John Wiley & Sons, 2007.
9. 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.
10. O. Bussmann. *The Future of Finance: FinTech, Tech Disruption, and Orchestrating Innovation*, pages 473–486. Springer International Publishing, Cham, 2017.
11. V. Butterin. A next-generation smart contract and decentralized application platform, 2014.
12. C. Cachin. Architecture of the hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.
13. C. Dannen. *Solidity Programming*, pages 69–88. Apress, Berkeley, CA, 2017.
14. N. Emmadi and H. Narumanchi. Reinforcing immutability of permissioned blockchains with keyless signatures' infrastructure. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, ICDCN '17, pages 46:1–46:6, New York, NY, USA, 2017. ACM.
15. R. Eshuis, A. Norta, O. Kopp, and E. Pitkanen. Service outsourcing with process views. *IEEE Transactions on Services Computing*, 99(PrePrints):1, 2013.
16. R. Eshuis, A. Norta, and R. Roulaux. Evolving process views. *Information and Software Technology*, 80:20 – 35, 2016.
17. Rik Eshuis, Alex Norta, Oliver Kopp, and Esa Pitkanen. Service outsourcing with process views. *IEEE Transactions on Services Computing*, 2014. In press. Preprint at `http://is.ieis.tue.nl/staff/heshuis/TSC2014.pdf`.
18. 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.
19. P.A. Hamburger. The development of the nineteenth-century consensus theory of contract. *Law and History Review*, 7(2):241–329, 10 2011.
20. 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.

Archived copy · cryp2nova.com

21. 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.
22. 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.
23. M. Kõlvart, M. Poola, and A. Rull. Smart contracts. In *The Future of Law and eTechnologies*, pages 133–147. Springer, 2016.
24. L. Kutvonen, A. Norta, and S. Ruohomaa. Inter-enterprise business transaction management in open service ecosystems. In *Enterprise Distributed Object Computing Conference (EDOC), 2012 IEEE 16th International*, pages 31–40. IEEE, 2012.
25. 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.
26. L. Masinter, T. Berners-Lee, and R.T. Fielding. Uniform resource identifier (uri): Generic syntax. 2005.
27. D.L. McGuinness, F. Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.
28. Business Process Model. Notation (bpmn) version 2.0. *Object Management Group specification*, 2011. `http://www.bpmn.org`.
29. 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.
30. M.A. Musen. The protégé project: A look back and a look forward. *AI matters*, 1(4):4–12, 2015.
31. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1(2012):28, 2008.
32. N.C. Narendra, A. Norta, M. Mahunnah, L. Ma, and F.M. Maggi. Sound conflict management and resolution for virtual-enterprise collaborations. *Service Oriented Computing and Applications*, 10(3):233–251, 2016.
33. A. Norta. *Exploring Dynamic Inter-Organizational Business Process Collaboration*. PhD thesis, Technology University Eindhoven, Department of Information Systems, 2007.
34. A. Norta. *Creation of Smart-Contracting Collaborations for Decentralized Autonomous Organizations*, pages 3–17. Springer International Publishing, Cham, 2015.
35. A. Norta. *Establishing Distributed Governance Infrastructures for Enacting Cross-Organization Collaborations*, pages 24–35. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.
36. A. Norta and P. Grefen. Discovering Patterns for Inter-Organizational Business Collaboration. *International Journal of Cooperative Information Systems (IJCIS)*, 16:507 – 544, 2007.
37. A. Norta, P. Grefen, and N.C Narendra. A reference architecture for managing dynamic inter-organizational business processes. *Data & Knowledge Engineering*, 91(0):52 – 89, 2014.
38. A. Norta and L. Kutvonen. A cloud hub for brokering business processes as a service: A "rendezvous" platform that supports semi-automated background checked partner discovery for cross-enterprise collaboration. In *SRII Global Conference (SRII), 2012 Annual*, pages 293–302, July 2012.

Archived copy · cryp2nova.com

39. 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.
40. A. Norta, K. Nyman-Metcalf, A.B. Othman, and A. Rull. âĂIJmy agent will not let me talk to the generalâĂİ: Software agents as a tool against internet scams. In *The Future of Law and eTechnologies*, pages 11–44. Springer, 2016.
41. A. Norta, A. B. Othman, and K. Taveter. Conflict-resolution lifecycles for governed decentralized autonomous organization collaboration. In *Proceedings of the 2015 2Nd International Conference on Electronic Governance and Open Society: Challenges in Eurasia*, EGOSE '15, pages 244–257, New York, NY, USA, 2015. ACM.
42. Aafaf Ouaddah, Anas Abou Elkalam, and Abdellah Ait Ouahman. *Towards a Novel Privacy-Preserving Access Control Model Based on Blockchain Technology in IoT*, pages 523–533. Springer International Publishing, Cham, 2017.
43. R. Ragunathan, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: The next computing revolution. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 731–736, New York, NY, USA, 2010. ACM.
44. T. Roxenhall and P. Ghauri. Use of the written contract in long-lasting business relationships. *Industrial Marketing Management*, 33(3):261 – 268, 2004.
45. A. Rull, E. Täks, and A. Norta. Towards software-agent enhanced privacy protection. In *Regulating eTechnologies in the European Union*, pages 73–94. Springer, 2014.
46. J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.
47. 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.
48. 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.
49. L. Sterling and K. Taveter. *The art of agent-oriented modeling*. MIT Press, 2009.
50. M. Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *International Workshop on Open Problems in Network Security*, pages 112–125. Springer, 2015.
51. M. Vukolić. *The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication*, pages 112–125. Springer International Publishing, Cham, 2016.
52. 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.
53. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.

Archived copy · cryp2nova.com