2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)

Beagle: A New Framework for Smart Contracts Taking Account of Law

Wei-Tek Tsai[1,4,5,6,7], Ning Ge[2], Jiaying Jiang[3], Kevin Feng[6], Juan He[1]

[1]Digital Society & Blockchain Laboratory, Beihang University, Beijing, China
[2]School of Software, Beihang University, Beijing, China
[3]Emory University School of Law, Atlanta, Georgia 30329, USA
[4]Arizona State University, Tempe, AZ 85287, USA
[5]Beijing Tiande Technologies, Beijing, China
[6]Andrew International Sandbox Institute, Qingdao, China
[7]IOB Laboratory, National Big Data Comprehensive Experimental Area, Guizhou, China

Abstract -- **This paper presents a new Beagle framework for Smart Contracts (SCs) taking account of law. Different from previous SC development or execution frameworks, this framework takes a practical approach to integrate law into SCs. Instead of translating legal contracts into codes directly, this paper proposes to treat SCs as a key component of legal contracts, use SCs to partially automate the executions of legal contracts, and produce legal evidence during the process. Thus, the proposed SC design will be significantly different from previous SC designs, not in programming languages to be used, but in the way SCs are designed and executed. This Beagle framework has five stages: SC template development from domain analysis, formal SC model and code development from templates, verification and validation (V&V), SC execution, and runtime monitoring.**

1. Introduction

The SC was proposed in 1994 by Nick Szabo as a set of promises, specified in the digital form, including protocols within which the parties perform on these promises [Szabo 1997]. SC has not been widely adopted and people are still exploring its potentials. Following the trend, this paper proposes a new framework for smart contracts. the new framework is unique from three aspects: legal aspects, infrastructure aspects, and design and execution aspects.

Legal Aspects. One of the most distinctive features of this new framework is that it takes law into consideration. Some of SC systems have been developed without consideration of law, e.g., Hyperledger has designed such a system called *chaincode*, i.e., code runs on top of a blockchain (BC). In this way, the system is purely technical, without any legal implications. This design is not the direction this paper will take, instead, this paper focuses on the integration of SC and law, using SCs to partially automate the executions of legal contracts, and produce valid evidence.

In fact, Hyperledger makes the right decision, a SC without legal meaning should not be called an SC. A SC is not a legal contract because a legal contract requires elements that cannot be met by the current SC technology, at least without changing the current law. For example, Potential parties need to negotiate contractual terms to establish a meeting of minds, which is a substantial requirement for contract formation. This step can hardly be fulfilled by SCs due to their mechanical and execution characteristics.

Another example is that, in most situations, legal contracts require signatures on important pages, but a SC has no place to sign. While digital signatures are accepted on legal contracts, but currently it is not possible to accept digital signatures by SCs.

However, SCs are still important if they interact with legal contracts in the future to partially carry out some legal tasks [Yu 2017a], e.g., contract execution. Section 3 presents our approaches to this aspect.

Infrastructure Aspects: Initially SCs have been proposed in the 1990s without consideration of BCs, and this SC concept remained dormant for many years. The SC concept receives attention when BCs such as Ethereum are deployed with SCs. Even though the DAO event caused numerous issues during 2016-2017, the event actually helped to push the development of SCs. Many new projects such as Kantara and OpenLaw have been initiated to further research and experiment on SCs. This is a critical aspect as a SC cannot be a valid SC if it does not run on top of BCs; Otherwise potentially data produced cannot be genuine. If data are not reliable, valid evidence cannot be guaranteed [Tsai 2019]. If not able to produce valid evidence, the SC execution is useless from the legal point of view. Section 4 discusses our approach in this aspect.

Design and Execution Aspects: This is commonly viewed as the SC computing aspects as this deal with design, implementation, verification and validation of SCs. This aspect was most extensively covered in the literature.

These three aspects are inter-related to each other. For example, each SC must have legal consideration, must run on top of a BC, and must be executable and the results can be validated. The execution results of SCs may be submitted to an arbitration court for the verdict.

This paper proposes a new framework Beagle for developing SCs. The name was inspired because Beagle a specific dog breed is often involved in law enforcement at airport customs. Similarly, SCs are involved in execution aspects of legal contracts. The

978-1-7281-1442-2/19/$31.00 ©2019 IEEE
DOI 10.1109/SOSE.2019.00028

framework is unique and divides SC development and execution into five stages:

(1) **Template Production**: This is the stage where SC templates are developed.
(2) **Formal SC Model and Code Development**: This is the stage where SC models and their code are developed.
(3) **Execution**: This is the stage where SCs are executed on the BC.
(4) **Verification & Validation (V&V)**: This is the stage where SCs are validated by users and verified by formal verification and testing techniques.
(5) **Runtime Monitoring**: This is the stage where SC execution is being monitored to ensure that execution follows the constraints specified in SCs.

This is an international project with American and Chinese participants. Emory University School of Law and Arizona State University are US partners, and Beihang University and other organizations are Chinese partners.

This paper is organized as follows: Section 2 covers related work; Section 3 addresses legal aspects of SCs; Section 4 presents the SC template production process; Section 5 discusses SC model and code generation from templates; Section 6 presents SC execution issues; Section 7 covers SC verification and validation (V&V); Section 8 covers runtime monitoring; and Section 9 concludes this paper.

2. Related Work

This section will briefly introduce five SC projects, followed a short evaluation.

Ethereum SCs: SCs did not get attention until Ethereum came with programmable SCs on BCs. Ethereum supports programmable SCs and provides a virtual machine EVM to execute SCs deployed [Buterin 2014, Wood 2014]. Ethereum provides a Turing-complete programming language to support engineers to develop SC code. It is like the Apple Store, where everyone can create and sell applications to users. Similarly, everyone can create their SCs and these SCs can be made available for others.

However, in the Ethereum, when an SC completes its execution, the results do not need to go through the consensus process before they will be written into the database. Thus, potentially it is possible that SC execution may produce different results and they get saved in the system.

Hyperledger SC: Hyperledger Fabric [Androulaki 2018] is an open-source BC system developed by Linux Foundation with IBM as the lead. It is a permissioned BC without resident tokens. It uses the term "chaincode" instead of SC because like Ethereum the chaincode has nothing to do with legal contracts.

A distinct feature of this system is that the ledger and the chaincode parts are separated, and chaincode execution results must be voted before they can be placed into the ledger. Thus, it is easier to manage chaincode execution than other SC systems, and the chaincode execution results are more likely to be correct as they have been voted. Specifically, it has two kinds of servers, *committers* and *endorsers,* committers vote on the results produced by endorsers, while endorsers perform the chaincode execution, but they are not involved in consensus voting. In this way, the chaincode and ledger aspects are separated in Hyperledger. Hyperledger also uses chaincode as a part of the consensus process for transaction validation.

Corda: Corda is a system inspired by BCs, but it is not a BC system. A Corda SC is an agreement that has two parts, the first is the executable part that can be executed and during execution accept human input. And the second part contains legal proses that involved parties need to comply. Corda SCs links business logic and business data to associated legal prose to ensure that the financial agreements on the platform are rooted in law and can be enforced. The Corda system has its unique consensus mechanisms, and not all the nodes participate in this process, running SCs on top Corda needs to fit into the Corda execution model.

OpenLaw: OpenLaw (https://openlaw.io/) is a BC platform to integrate legal contracts with SCs. It allows lawyers to model all or parts of legal agreements using SC to decrease the cost and friction of creating, securing, and generating binding legal agreements. It turns the traditional legal contract into documents with embedded SC code using Legal Markup Language similar to Wiki Text. Lawyers can digitally sign and securely store legal contracts using OpenLaw - while maintaining "user-friendliness" and industry compliance.

However, most contracts today use natural languages, and these languages are significantly different from programming languages that SC code will be expressed. Thus, the following problem is that it is difficult to translate an existing legal agreement into a SC as these two are significantly different matters. One term in natural language might have multiple explanations but a programming language can have only one semantic (otherwise an execution of the code on different machine may produce different results), and it is difficult to encode all the explanations in programming languages. Even if encoded, it is hard to decide which one should be executed in a specific context. The law is generally artificial, local, and uncertain, while public BCs are often automated, global, and deterministic. The two are closely related and maybe in conflict.

Kantara Initiative: This is a BC and SC discussion group (BSC DG) to discuss various issues after the DAO event. They propose that SCs must have the following features [Hardjono 2017]:

Meaningful programmatic code: The code must perform meaningful action involving the named subjects and objects.

The digital representation of real-world subjects of the agreement: The legal parties involved must be validly represented digitally within the code. This feature requires digital identities.

Digital representation real-world objects and/or actions of the transaction: The legal objects (e.g., assets) involved must be validly represented digitally within the code.

Verifiable correspondence between actions represented in code and actions in the real world: The actions represented by the code must correspond to real-world actions or changes of state recognized within the given legal context/domain.

Legal prose meaningful within the designated legal context/domain: Legal prose – understandable to actors within the legal domain – must accompany and be bound to the code portion (e.g., digitally signed).

3. SC Legal Consideration

The new SC framework will integrate with law in two ways. First, SC should be designed to partially automate the execution steps in legal contracts. Second, SC execution should produce data that can be used as legally valid evidence.

3.1 Automating the Contract Execution

Contract execution is the process whereby the contractual parties perform their duties according to the legal agreement. Contract execution can be done both online and offline, depending on the nature of the contract and the contractual terms. For example, the execution of a sales contract – purchasing fruits in the market – should be done by physical actions. The execution process is the seller hands over fruits, then the buyer pays money. If a transaction occurs online, then the execution of the sales contract becomes technology-driven processes: the buyer placing her order by clicking payment, banks transferring money from the buyer's account to the seller's account, the seller accepting the order and transferring the ownership of items. SCs can automate online processes, but not offline conducts. Therefore, some SCs may partially, not fully, automate the execution of legal contracts, particularly those contracts with offline activities.

The concept of partially automating the execution of legal contracts is different from the ideas of existing SC projects. For instance, comparing to OpenLaw with a focus on translating all or parts of legal contracts to code, this Beagle project aims at executing legal contracts. OpenLaw acts as a translation tool, and it carries no legal implications. In contrast, our SC concept involves legal implications. It is part of the contract execution processes. It carries not only letters of the law but also bears legal consequences.

3.2. Producing Valid Evidence.

Evidence is something legally submitted to a tribunal to ascertain the truth of a matter. It determines what information can be presented in a legal proceeding. Evidence can come in a wide variety of forms, such as a piece of writing, a fingerprint, a testimony, a picture, a video, and a set of experimental data. But due to nature of SCs, this paper refers to online materials, presented in the form of data [Yu 2017b].

Without considering excision and exception rules, to be valid, evidence should bear three properties: relevance, truth, and legality.

Relevance means that evidence is relevant if it has any tendency to make a fact more or less probable than it would be without the evidence. To be relevant, the specific piece of evidence must relate to some time, event, or person in the present lawsuit.

Truth means the evidence presented should be authentic. What evidence tends to prove should be true and objectively exist, because any case would occur both in space and in time. What happened was subject, not object. Law requires that all materials should be proved to be true to be valid.

Legality has three implications. First, evidence should be collected by legal authorities through legal procedures and methods; second, evidence should have legal forms; third, evidence should come from legal sources.

SCs could perfectly produce valid evidence that bears three properties: relevance, truth, and legality. As mentioned, this paper narrows the scope of evidence to online data. SCs can fulfill the requirements of valid evidence via producing **real-time**, **process**, **and immutable data.**

Real-time data: The data must be collected in real time or near real time for most IT applications. As data can be easily changed by IT systems, data not collected in real time may have been changed before they are entered into BCs. SCs can produce real-time data as they run on BCs. Once data are produced in a real-time basis, they prove the relevance property of the evidence, because the existence of such data has a tendency to make a fact more or less probable than it would be without the data. And it also helps to determine the truth of evidence as data record what happened in time and space objectively.

Furthermore, data collected should have associated data such as time of the event, data collection agents or devices and their IDs, other relevant information such as the communication medium used to transmit the data from the source to the BC. For example, an event that a person entered into a private room, the time of the event, the associated photo, the device that captured the photo, the communication device and wire used to transmit the data are all relevant data.

Process data: It is necessary to collect data during the process of the event, not just the result

data [Yu 2017b]. The reason that SCs can collect both the process data and the result data come from a BC. A complete record of data reflects the complete story of an event or a transaction. This also elevates the relevance and truth of data and ascertain evidence's validity. It can also support the legality of the evidence produced.

In the future, it is possible that the law will allow validated SCs, such as by proper validation agencies, can be a valid source form of producing evidence, and the data on SCs are a valid form of evidence. Additionally, gathering evidence through SCs would be a better procedure and method than collecting evidence by the human as SCs are more technical and objective, with less human mistakes.

Immutable data: Data collected must be preserved and have not been modified. SCs that run on top of a BC can support this feature. The BC can guarantee the immutability through its cryptography and consensus mechanisms. The immutability is the most significant and efficient proof that evidence is authentic and trustworthy. It also tends to make a fact more or less probable than it would be without the evidence. In other words, immutable data also demonstrate the relevance property of valid evidence.

4. SC Template Production

Instead of translating legal contracts into SCs, the Beagle framework suggests that domain analysis should be performed to develop SC templates that can be used to develop SCs later. The template will cover those common terms and conditions in an application domain, e.g., real-estate transactions. While each individual real-estate transaction is different, but most of these transactions share significant commonality. The goal of templates to capture these commonalities, furthermore, significant effort is made to these templates so that they can be easily translated into SC code once they are substantiated.

4.1. Design Principles of SC Templates

This section defines a set of principles for the design of SC templates regarding the legal agreement process. The design of SC templates needs to follow the following six principles: process-based principle, trusteeship, consensus, oracle, accountability, and rollback. One can see that this work is essentially *domain analysis* in software engineering, and this time the goal is to generate a set of SC templates for a specific application in a domain. The template concept carries two important goals: 1) common legal processes for that particular application will be captured including not only normal scenarios but also failure scenarios; 2) the template will be stated in a way that facilitate formal model and code generation later.

This section takes a real-estate purchase contract as an example to demonstrate SC templates. The lifecycle of a contract includes contract formation, execution, enforcement, and remedies, see Figure 1. SC templates will be applied to the execution phase of a contract, focusing on the buyer and seller's main legal obligations, i.e., the exchange of consideration and ownership.

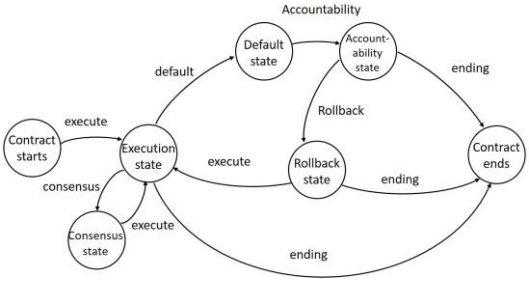

Figure 1: State Diagram of Contract Lifecycle

Process-based principle: We asked lawyers to label the main rights and obligations in each type of contracts. These rights and obligations will be execution objects of a SC. In addition, lawyers also need to estimate the potential disputes at every step of the contract execution and come out with possible solutions that can be automatically executed. The "disputes-and-solutions" process will be designed accordingly in SC templates. The Beagle abstracts the logic of the process into a contract template with configurable variabilities.

As contracts have many categories, SC templates also have many categories. For a practical system, it is necessary to develop a library of SC templates for a variety of applications. Each template contains a main legal process (the main rights and obligations) to enforce a set of variable processes.

Taking the real-estate purchase contract as an example, the main rights and obligations are that the buyer transfers consideration and the seller transfers the ownership of the property. Thus, exchanging the consideration and the ownership of the property is the legal process to enforce in the template.

If something goes wrong, e.g., the payment by the buyer not fulfilled by the due date, the template will specify steps to stop the transfer of the ownership. This is how "disputes-and-solutions" pattern can be designed in SC templates.

Trusteeship Principle: In today's business, sellers and buyers for a transaction will request a law firm to be the trust to ensure that payments are made, and assets are accurate and legal. In cryptocurrency, this mechanism is not used [Tsai 2018a, Wang 2018, Bai 2019]. However, trusteeship will still be needed if other kinds of assets involved, e.g., real estate, stocks, or bonds. The trusteeship process is divided into two main steps:

Step 1 (confirmation stage): An agency receives the money transferred from the buyer, and the commodity information from the seller, then verifies the payment from the bank, and the

authenticity of the information from the government office.

Step 2 (transfer stage): Next, the trustee agency will transfer the payment to the seller and charge a fee. All the above steps concerning the trusteeship process are performed on SCs. A trusteeship is modeled as a main process in a Beagle template, see Figure 2.

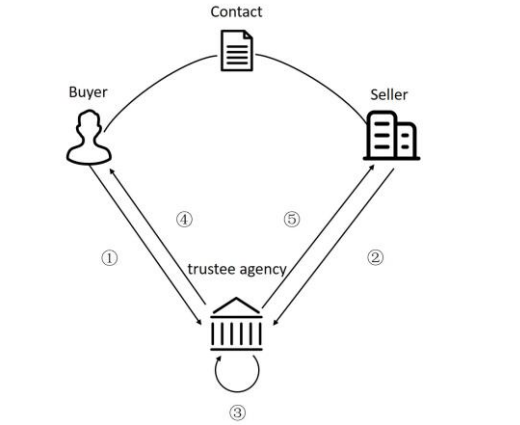

Figure 2: Trusteeship Process

Once the main processes are specified, irregular processes can be identified. For example, payment not made, and assets not real are two examples that result in process termination without the transfer of the ownership of an asset.

Oracle Principle: The Beagle model requires all the data needed for SC execution come from the underlying BC, and any resulting data must be written into the BC. However, not all data will be produced by SCs, and thus some data may come from outside of the BC, such as the Internet. The Beagle model requires that any data outside of a BC must go through an oracle process to ensure that data entered are correct.

Data may still be incorrect even if data have gone through the oral process. Reasons are various, such as failure in communication, synchronization faults, or malicious attacks. Thus, data in BCs should integrate an integrity score system. High integrity-scored data means that data are more likely to be correct, and low integrity-scored data are prone to be wrong [Tsai 2018a, Tsai 2018b]. When specific data entered into a BC, the source of data, as well as any existing integrity scores will be recorded. For example, if the data come from another BC with existing integrity scores, the scores will be recorded and adjusted for the integrity level of the BC. If the BC has high integrity ranking, the same score can be used. If the BC has a low ranking, the integrity score will be lowered accordingly. The BC may also rank the integrity level of any incoming data source to provide the initial integrity score for the data from that particular source.

Integrity computation will follow the Biba integrity model, i.e., users can create content only at or below their own integrity level. Thus, if a BC has B integrity ranking, all the data coming from that BC can have at most B integrity score.

For example, if the BC uses two data to perform a computation, the resulting data will have the lowest integrity score of the input data.

As the integrity scores of data will in general go down in the system, the BC may employ Integrity Evaluators (IEs) to raise integrity scores from time to time. These IEs uses domain application rules to raise data integrity scores. For example, if data are financial data, accounting principles may be used to verify that the data are consistent with other data, and if they are consistent, the integrity score can be raised. For example, *totalAmount* can be determined to be *shareNumber* multiplied by *sharePrice*. If the *sharePrice* and *shareNumbe*r are known to be of high integrity, and *totalAmount* is consistent with these two data, totalAmount integrity score can be raised to be the minimum of *sharePrice* and *shareNumber*. In this case, integrity scores can be maintained in BCs.

Consensus Principle: When executing a SC, each participating node on the BC computes data independently. These nodes are expected to produce consistent computing results. If more than one results are obtained, the BC will check which one is the correct result. This process is used by Hyperledger [Androulaki 2018].

A template will identify key events in the corresponding legal contracts. In a real-estate transaction, key events are where initial purchase agreement signed with the right deposit, proof-of-ownership obtained, inspection report obtained, any amendments completed, total payment made, and ownership transferred. Evidence of these key events must go through the consensus process of the BC to ensure that all the parties, such as buyers, sellers, loan bank, title agency received the same information at this round of consensus voting. Each of these key events must be fulfilled before a complete SC is done. The process may take seconds, days, weeks or even months.

During the process, data are stored in the BC as intermediate data. They cannot be modified due to the BC immutability characteristic. If data entered are incorrect, a user may request a new item to be added into the BC without changing the existing data. In this way, the new and corrected data can be incorporated while not compromising the BC immutability property.

Data entered also have two timestamps. One is the time when the data is recorded by relevant agents. Another one is the time when the data is entered into the BC. Two timestamps will be useful

in validating the data in case of court proceedings. Each time any data entered into the BC, the data must have gone through the consensus mechanisms of the BC.

BC may perform check on all or selected sample data to ensure that data entered are correct through the consensus process. If data are found to be inconsistent, this may signal the BC is compromised as data entered must have gone through the consensus process earlier. If any node in a BC has any inconsistent data, the particular node may have been compromised. Figure 3 illustrates a long process.

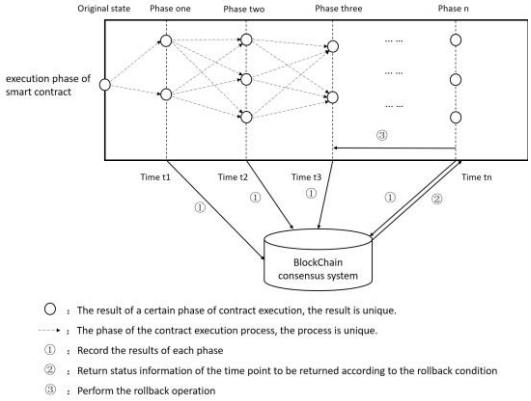

Figure 3: A SC with a Long-Lasting Process

Accountability Principle: The design of SC templates should follow an accountability principle. Assuming the transaction of SC is not fulfilled at some stages, who should be responsible? The Beagle framework applies the accountability principle to address this issue. In the template, a preliminary judgement as to whose fault should be made, followed by a termination or recovery of the transaction.

For example, in a real-estate purchase contract, if the final transaction is not fulfilled, the one who is at fault should bear consequences accordingly. If it is the buyer's fault, i.e., purchasing a house using an illegal source of money. then the buyer should bear the responsibility that the process would be suspended, thus the buyer cannot get the ownership of the house.

Rollback Principle: A SC template not only describes correct and smooth process, but also deal with problematic process. The way a SC template reacts to the problematic process is the rollback process. For example, if *initialDeposit* to the trust company is made, the SC will record that the deposit is available. However, a week later, the trust company was informed that the source of the deposit is not proper, such as the money was borrowed from another party, the SC should roll back the process to the point where the *initialDeposit* not done.

This can be done automatically using event analysis such as event tree analysis. Combinatorial analysis of related events can be analyzed and stored in the template to ensure that the template can address all the possible event sequences. This is important because the goal of Beagle SC is to make these SCs as a part of legal contracts, and legal contracts needs to address any possible sequences of events.

5. SC Model and Code Generation

This phase develops SC code from templates. In this project, the SC has three components:

- NL contract (NL is an abbreviation of Natural Language);
- Formal SC model;
- Executable code.

This process is divided into two stages: development of formal SC model, and development of SC code from the formal model.

Development of Formal SC Model

Users (lawyers and contract participants) first select a SC template and configure the contract template. Parameters like deadlines and the price of real estate are determined and filled by users. Once the configuration is done, the formal SC model is created. This will serve as the second component of the SC. The relationships between the formal model and NL contract will be maintained, e.g., an item in the formal model may correspond to the corresponding item in the NL contract. Thus, if the formal model or NL contract is changed, the corresponding change will be updated accordingly.

At the same time, users need to check the agreement in the textual contract updated as user inputs. When finishing configuring the template, a SC model is built from the system. Users can get legal regulations through the search engine or recommendation engine, then input the contract parameters.

Development of SC Code from Formal SC Model

SC code is generated from the formal SC model to run on the SC platform. The generation can be automated, but it is difficult to automate the generation process completely.

This project also generates interfaces to interact with external SCs. To facilitate the code integration, Beagle code generator also generates API of SCs.

SCs often involve financial transactions of digital assets. It must be highly reliable and secure [Bai 2019]. Thus, its development needs to follow rigorous development methodology to prevent any potential faults in code or in the model. For this reason, the Beagle framework follows the Model-Driven Engineering (MDE). MDE focuses on develop correct software with a set of formal modeling and verification techniques, successfully applied to developing safety-critical systems [Ge 2017, Ge 2018a, Ge 2018b].

In addition to formal development, MDE used must also cover the legal aspects. SCs can be lawful

if the legal experts confirm that the SC code is consistent with a legal contract with respect to both the process and data involved. However, similar to the cost of drafting a paper-based contract, the development of the corresponding SC is also a costly and time-consuming process. The MDE supports two important features: rapid iterative development and friendly human communication. The MDE supports rapid iterative development by maintaining traceability between contract templates, between SC model and code, by automating model translation and code generation. The MDE supports friendly human interface by providing quick and clear feedback to developers, lawyers and participants in user-friendly manner thanks to formal model. For example, if the buyer transfers consideration but the seller does not transfer the property ownership, a visual scenario (called counterexample) will be automatically generated by simulator to help users to understand the failure.

6. Execution

This phase executes SC code at runtime. There are several issues in this phase described below.

Triggering: Which system can trigger the execution of a SC? Multiple choices are available: 1) triggered by the change of data in the BC; 2) triggered by the application process

In the first case, the SC infrastructure design can be complex. Specifically, if the system has lots of SCs with numerous conditions attached, the system needs to capture these conditions that are satisfied in real time to trigger the SC execution. One way to address this problem is to have a set of dedicated servers to watch these conditions. They need to reach a consensus to trigger an SC execution. In this way, each SC execution will start at the same time with all the participating nodes. But this will slow the SC process and add complexity to the SC infrastructure.

In the second case, as multiple voting servers are active in the BC, is it a good idea that only one server will trigger the execution. However, this only sever needs to be selected dynamically, which can be the single point of failure. This issue can be addressed by having a circular chain of nodes. Each node watches over the previous node in case of failure. Any failure will be automatically taken care by the following node in the chain, but the chain position needs to be decided each time.

A static choice will not be an acceptable solution as what happens if the statically chosen node is compromised. Another way is to have a voting consensus process to decide if an SC is needed. For mission-critical applications, a 100% vote may be necessary before the SC can be triggered, e.g., a large transaction.

Data used by SC execution: In the Beagle model, only data stored in the BC can be used by SCs. This requirement is to ensure only correct data (for legal purpose) are using as data stored by the BC are assumed to have higher integrity.

External data can be used only after it has been accepted by the BC Oracle and consensus voting process. However, this introduces new problems. As a SC can be a long-lasting process, can the system use those new data entered into the BC in the SC or not. For example, any data that arrive after the SC is triggered will be considered as *new* data. An SC can distinguish these new data from old data by examining the timestamps of associated data.

One way to address this problem is that the triggering function specify the timestamp of data to be used. In this case, even the data are changed during the SC execution, the right data have been pre-specified.

However, this may not be the optimal solution. In the real estate purchase contract, an SC may prefer to use any new data rather than using the existing data. For example, amount paid may be minimal in the beginning, but as buyers keep on depositing money, the balance will increase, and the SC should use the new data. *HasHouseInspected* is another item that an SC prefer to use data. In the beginning, the item carries *No*, and later must be changed to *Yes* before the transaction can go through per state regulation.

However, not all the data can be changed, for example, buyers, sellers, house information should not be changed, in fact, if they are changed, this may indicate a potential fraud. For example, if seller information is changed, this may indicate that the seller may not be the legal owner.

Thus, a better solution is to mark those items that should take new data as they come into the BC, and those items that should not have any new data during the process. When an item that should not have any new information has new information, the SC will automatically trigger a process to terminate the transaction.

Number of execution processes: One can design a SC that is executed only by one server. This will cause the server to be a single point of failure. The other extreme is that the SC is run on top of all voting servers or selected participating servers. Or one can design a system that selected servers are involved in SC computation.

Execution servers: Another issue is that whether these execution servers can be logically or physically the same as voting servers in the BC. In some BC design, they can be logically different, but physically they can be the same depending on system configuration.

Completion of SC execution: Once a SC has competed its execution, and the results have been accepted by the BC consensus process, all the data produced by the SC must be written into the BC. This has three implications:

1) Resulting data cannot be saved at any place outside of the BC until it is saved at the BC first;

2) Resulting data must be saved at the same block positions with the same consensus voting;

3) A BC can store two kinds of data only. First is the data produced by SCs; second is the application data with high integrity, and additionally an oracle process has also verified the data.

These three principles maximize the probability that all the data stored in BCs are correct.

7. Verification & Validation

As software code, the correctness of SC needs to be checked by V&V. However, SC is different from traditional software in numerous ways. Accordingly, their V&V process and methods are different from traditional V&V processes.

The new framework for SCs should allow the developers to verify and validate their implementation. This section discusses five important issues related to the V&V of SCs:

- What are the differences between SC software and traditional software?
- What are the sources of SC faults?
- What are the SC desirable properties?
- In which stages of SC development should perform V&V by using what methods and tools?
- How can one apply crowdsourcing to SC V&V?

7.1. SC and Traditional Software

SC software is different from traditional software in various aspects including data immutability, process reversibility, attack availability, and man-in-the-loop.

Data immutability: As discussed in Section 3.2, data produced from SCs are preserved in BCs. This provides traceability, but data analysis become complex and involved due to unique BC data structure.

Process reversibility: In cryptocurrency, transactions cannot be reversed. But, in most countries, transactions can be reversed even few days after trade completion, e.g., a stock transaction can be reversed two days after the trade. To implement this rollback mechanism, SCs need to have pre-specified rollback mechanism for the system to return to the state. This also means new data must be added to the BC to indicate the previous data entered are no longer valid as the BC cannot allow any data change once entered. Another way to do this is to save those intermediate data as separate data and commit the final data after the settlement day is over. But this mechanism will introduce other issues as sometimes SCs can last for a long time such as months, intermediate data can be enormous in size.

Attack availability. A SC in a public BC can be accessed by all the nodes, and they can be attacked by anyone like the DAO event. A SC in a permissioned BC can also be attacked by participating nodes. This is the reason that SCs and their running platform are prone to be attacked. This vulnerability requires encoding solutions to handle attacks in SCs or in its running platform. It also requires the platform to monitor and verify the execution of SCs online.

Man-in-the-loop. Currently SCs complete transactions without third parties as in cryptocurrency systems, but this will not be true in financial systems. Under the current law, parties involved must sign legal documents before they become binding. Thus, development and execution of SCs without signatures under the current law will not be legally valid. Not only signatures, numerous transactions today require external legal documents not available in SCs, e.g., property ownership certificates. These can eventually be digitized too, e.g., they can be issued as digital certificates stored in BCs, but currently they are not, and thus current SCs must live under the current legal condition. These should be considered in performing V&V for SCs.

7.2. Fault Sources of SCs

An SC may fail due to many reasons such as:

- Transaction process not properly designed;
- SC code not properly implemented;
- SC platform not properly implemented;
- Communication between distributed BC nodes not properly synchronized;
- External attacks.

The first three types of faults originate from the software, while last two from the SC platform.

Faults in transaction processes might arise from the transaction process not properly defined, the wrong expression by the participants, improper legal interpretation and actions, and mistakes in legal contracts. Some of these issues can be addressed by the consensus process in BCs and SCs, and newly added rollback mechanisms.

Faults in SC implementation might come from SC design templates, formal or informal SC models, and SC code. These faults may come from human errors or by code or model generators.

Faults in platform implementation might come from the BC platform. For example, the DAO event was caused partially by the Ethereum platform.

Faults in communication systems are due to events and data synchronization problems in distributed systems. A pair of local ordered events might not preserve the order when they arrive at other distributed nodes, and this may trigger wrong events for an SC action. This is a serious problem as multiple (from few to thousands) transactions are grouped in blocks, and they are processed at the same time by BCs. This introduce new kinds of synchronization problems not encountered before.

Faults in execution environments come from the failure of hardware or the third-party software on the platform system, the human operation like wrong operations, or external attacks [Tsai 2017]. A survey of attacks on Ethereum SCs can be found in [Atzei 2017].

7.3. Desirable SC Properties

SCs, being critical processes, should cover the above fault sources. Historical attacks on SCs provide guidance to develop appropriate techniques.

Process correctness: The legal process in SCs should conform to the true intent with respect to the process as well as data, e.g., a SC should roll back when the termination conditions are satisfied, and the rollback mechanism should roll back to the appropriate time points in case of partial rollback.

User behavior property: Most legal SCs have the man-in-the-loop feature. Each user interaction needs to specify intended user behaviors at the appropriate process step with expected data. For example, the SC needs to check if proper payment has been made and expect the answer is *yes*. In case of *yes*, the process will move to the next step; in case of *no*, the process will either stop or roll back to the previous step.

7.4. V&V in SCs Development

At each step of the MDE, different formal verification methods can be applied. The SC V&V process can be divided into three stages:

- **Legal analysis**: This is performed by requirement modeling and validation method;
- **Contract development**: This is done by formal contract modeling, formal verification and formal code generation;
- **Contract execution**: This is done by runtime verification.

Today, most existing works perform testing and formal verification to the SC models and code. This aspect would not be addressed as such works do not interact with law.

Legal analysis by simulation

The template-based approach allows developers, lawyer, and participants to develop various transaction models. When users fill a SC template with data, an actual contract model is built. Even though legal analysis has been done in the template earlier, the actual contract model should be subject to legal analysis. The model is formal, and thus can be validated by running scenarios on the formal model. This process is a simulation process.

In the Beagle framework, interactive simulators help users validate the SC model to satisfy their intention. Lawyers and developers can rely on the simulation tool to accelerate the iterative development of SC templates and enrich the library of SC templates. The simulation tool can be enhanced each time a specific contract is analyzed as each contract will add new scenarios.

SC design verification by formal techniques

Simulation can help users validate the model but does not guarantee the correctness of the SC model because the search is not exhaustive. Since the birth of SC, significant works have been done on applying formal methods to SC code.

The Beagle framework uses MDE, and significant verification effort will be on the formal contract model. Model checking can be used to verify the contract model. As SCs reflect transactions, the structure is mainly composed of conditional branches. Most of the SCs have relatively low computational complexity, and the state space is usually finite, making them suitable for adopting automatic theorem proving (ATP) methods like model checking. A more complicated contract can still be verified by interactive theorem proving (ITP) methods under the premise if computing resource is limited. But this method may increase the manual effort and make the verification work semi-automated.

A project showed that it is feasible to verify the correctness and necessary properties of a SC template using SPIN model checker [Bai 2018]. This Beagle project chooses Timed Automata (TA) as the template modeling language because most legal processes have time constraints. Various model checkers such as UPPAAL can be used to verify SC models developed in TA.

SC Code Verification

Many recent works use model checking techniques to verify the Solidity contract code. For example, the work [Abdellatif 2018] verified SC code and BC execution protocol along with users' behaviors based on a BIP model checker. The work [Chen 2018] verified concurrency problems using Maude model checker. The work [Qu 2018] checked the vulnerability in SCs especially from the perspective of concurrency using the CSP theory and FDR model checker. The work [Nehai 2018] verified that the application implementation of SC complies with its specification using NuSMV model checker. The work [Alt 2018] verified the functional correctness of SC code using SMT solver. Other works relied on theorem proof to verify the intended behavior of Solidity contract code [Bhargavan 2016, Amani 2018, Le 2018].

Although these works show feasibility to verify Solidity SC code, the field is still at its infancy with many issues:

Turing-complete language issue: The choice of Turing-complete language limits the possibility of thorough verification. It is expected that non-Turing complete language can overcome this hurdle [Atzei 2017]. Some works proposed experimental languages for this reason.

Property-completeness issue: As discussed in Sections 6.2 and 6.3, one needs to understand the sources, types, and effects of faults and define an SC property type system that allows developers to specify a complete set of properties.

State space explosion issue: Model checking techniques consume significant resources when the behavior of the target system is complex.

High learning curve issue: Formal verification techniques are costly to apply in real systems because the learning curve is high.

The Beagle will choose a language that is not Turing complete to reduce the design and verification effort. Most of the SCs are designed for financial transactions, thus being Turing complete for the SC model or language is not necessary. The model can use consistency tool to ensure the conformance between the design model and the generated code.

By using a template-based and process-based approach, the issue of state space explosion can be relieved by introducing property-specific state space reduction techniques.

Crowdsourcing property modeling (Section 6.5) and verification platform can be a promising direction to address issues in this Section.

7.5. Crowdsourcing Testing for SCs

This section applies crowdsourcing techniques for SCs testing.

Crowdsourced testing is an emerging trend in software testing. It makes use of the benefits, effectiveness, and efficiency of crowdsourcing and the cloud platform. Crowdsourced software testing has the advantage of recruiting, not only professional testers, but also end users to support the testing tasks. It has been applied to various types of testing activities, including usability testing, performance testing, GUI testing, test case generation, and the oracle problem.

As SC development is complex, crowdsourcing is a way to reduce development risks. As more people involved in development, a higher quality of testing will be reached. However, as SCs involve multiple domains of expertise, crowdsourcing needs to engage experts with different skills and knowledge.

Participants: Users, legal experts, software developers, formal method specialists, and software test engineers may work together to evaluate various aspects of SCs. They are encouraged to discuss with each other in social media or crowdsourcing platform to create a synergy of ideas and discuss potential strategies. Participants can gather externally and internally. External participants participate can work on funding. Internal participants can evaluate the results obtained from crowdsourcing, and suggest directions for further evaluation.

Processes: Crowdsourcing tasks need to be planned, organized, and even optimized. For example, specific tasks can be crowdsourced first, and the results can be evaluated by another crowd or by an internal team. In this way, crowdsourced tasks can be repeatedly performed, with the next task planned based on the results of previous tasks. These tasks can be performed concurrently or sequentially by external and internal teams. For example, SC templates can be evaluated by external crowd lawyers and by an internal team of lawyers. The internal team will determine the final product after several iterations. In this way, templates developed are more likely to be comprehensive and correct. These lawyers are free to use any tools and discuss with fellow lawyers.

Platform: It is better to conduct crowdsourcing tasks using a platform. The platform can support communication, act as an search engine, or serve to automate evaluation tools, such as formal method tools and event-tree analysis tools.

8. Runtime Verification

Even the model and code are formally verified to be correct, it is still necessary to monitor the runtime behavior and verify this behavior at runtime or offline.

For example, a SC is to buy IBM stock when the price hits $200. When the SC executes, can one have an independent evaluation that the IBM stock price has indeed hit the price specified? This can be done if the SC has another associated process that will automatically send a message to the SC user that the stock price has hit the target.

It is possible that the SC may still fail to complete due to the market condition. The order cannot be fulfilled even though the triggering condition has been met. Those monitoring processes are not SCs, and they do not need to follow the formal process of the Beagle, but they provide additional assurance to users that the right SC has been triggered.

This can be done for examples as follows:

- Whenever a SC is triggered, a message containing the SC ID, time of triggering, input data, and participating node ID will be sent to relevant users. These messages will be stored in BCs;
- Completion of the SC execution will generate another message containing SC ID, the results of execution, and time of the event to relevant users. These messages will be stored in BCs;
- Completion of SC, i.e., results are stored in BCs, will generate another message indicating the SC ID, results, block positions that contain the results, and ID of these blocks and nodes. These messages will be stored in BCs.

The work [Ellul 2018] showed how runtime verification techniques can be used in the domain of SCs. Although BC is different from traditional distributed software, runtime monitoring and verification methods can be applied.

9. Conclusion

This paper discusses fundamental issues of SCs where law is relevant. Once SCs consider the legal aspects, SC design need to make significant changes, and most of changes will be needed in the SC development and infrastructure. However, currently legal contracts are too far away from being the basis for formal legal computing as both

languages used and more importantly the structure and flow in these two are significantly different

Thus this paper proposes the Beagle framework to address these issues. From the perspective of law, this framework treats SCs as a key component of legal contracts, using SCs to partially automate the executions of legal contracts, and produce legal evidence. From the perspective of software engineering, this framework covers the chain of production, execution, V&V, and runtime monitoring with a template-based approach. The template is developed base on domain analysis for a specific application.

Formal modeling and verification are employed. In this Beagle framework, law and code are connected, code is generated from formal contract model, and the model is based on templates, and templates are based on legal regulations. In this way, legal rules and regulation can be executed eventually as a part of SCs running on top of BCs.

Additionally, this project is a collaboration projects between legal experts and computer scientists in two countries. It facilitates the understanding of two fields and advances cross-interdisciplinary studies. It provides a foundation for the convergence of the legal world and the computer science world.

10. Acknowledgement

This work is supported by National Key Laboratory of Software Environment at Beihang University, National 973 Program (Grant No. 2013CB329601) and National Natural Science Foundation of China (Grant No. 61472032).

11. References

[Abdellatif 2018] T. Abdellatif and K.-L. Brousmiche, "Formal verification of smart contracts based on users and blockchain behaviors models," in New Technologies, Mobility and Security (NTMS), 2018 9th IFIP International Conference on. IEEE, 2018, pp. 1–5.

[Amani 2018] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in isabelle/hol," in Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. ACM, 2018, pp. 66–77.

[Alt 2018] L. Alt and C. Reitwiessner, "Smt-based verification of solidity smart contracts," in International Symposium on Leveraging Applications of Formal Methods. Springer, 2018, pp. 376–388.

[Androulaki 2018] E. Androulaki, A. Barger, V. Bortnikov,C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich et al.,"Hyperledger fabric: a distributed operating system for permissioned blockchains," in Proceedings of the Thirteenth EuroSys Conference.ACM, 2018, p. 30.

[Atzei 2017] N. Atzei, M. Bartoletti, T. Cimoli. "A Survey of Attacks on Ethereum Smart Contracts (SoK)," in Maffei M., Ryan M. (eds) Principles of Security and Trust. POST 2017. Lecture Notes in Computer Science, vol 10204. Springer, Berlin, Heidelberg.

[Bai 2018] X. Bai, Z. Cheng, Z. Duan, and K. Hu, "Formal modeling and verification of smart contracts," in Proceedings of the 2018 7th International Conference on Software and Computer Applications. ACM, 2018, pp. 322–326.

[Bai 2019] X. Bai, W. T. Tsai, X. Jiang, "Blockchain Design -- A PFMI Viewpoint" to appear in 2019.

[Bhargavan 2016] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi,G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T Sibut-Pinote,N. Swamy et al., "Formal verification of smart contracts: Short paper,"in Proceedings of the 2016 ACM Workshop on Programming Languagesand Analysis for Security. ACM, 2016, pp. 91–96.

[Buterin 2014] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2014 accessed: 2016-08-22. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[Chen 2018] X. Chen, D. Park, and G. Ro ̧su, "A language-independent approach to smart contract verification," in International Symposium on Leveraging Applications of Formal Methods. Springer, 2018, pp. 405–413.

[Ellul 2018] J. Ellul and G. J. Pace, "Runtime verification of ethereum smart contracts,"in 2018 14th European Dependable Computing Conference EDCC) IEEE, 2018, pp. 158–163.

[Ge 2017] N. Ge, A. Dieumegard, E. Jenn, B. d'Ausbourg, Y. Aït Ameur, "Formal development process of safety-critical embedded human machine interface systems" in 11th International Symposium on Theoretical Aspects of Software Engineering, 2017, pp. 1-8.

[Ge 2018a] N. Ge, A. Dieumegard, E. Jenn, and L. Voisin. "Correct-by-construction specification to verified code". Journal of Software: Evolution and Process. 2018 Oct;30(10):e1959.

[Ge 2018b] N. Ge, E. Jenn, N. Breton and Y. Fonteneau. "Integrated formal verification of safety-critical software". International Journal on Software Tools for Technology Transfer. 2018 Aug 1;20(4):423-40.

[Hardjono 2017] T. Hardjono, and E. Maler, "Report from the Blockchain and Smart Contracts Discussion Group to the Kantara Initiative," June. 05, 2017. [Online]. Available:

https://kantarainitiative.org/file-downloads/report-from-the-blockchain-and-smart-contracts-discussion-group-to-the-kantara-initiative-v1/

[Le 2018] T. C. Le, L. Xu, L. Chen, and W. Shi, "Proving conditional termination for smart contracts," in Proceedings of the 2nd ACM Workshop on Blockchains, Cryptocurrencies, and Contracts. ACM, 2018, pp. 57–59.

[Nehai 2018] Z. Nehai, P.-Y. Piriou, and F. Daumas "Model-checking of smart contracts," in IEEE International Conference on Blockchain, Halifax, Canada,2018

[Qu 2018] M. Qu, X. Huang, X. Chen, Y. Wang, X. Ma, and D. Liu, "Formal verification of smart contracts from the perspective of concurrency," in International Conference on Smart Blockchain. Springer, 2018, pp. 32–43.

[Szabo 1997] N. Szabo, "Smart contracts: building blocks for digital markets," EXTROPY: The Journal of Transhumanist Thought,(16), 1996.

[Tsai 2017] W. T. Tsai, X. Bai, and L. Yu, "Design issues in permissioned blockchains for trusted computing." 2017 IEEE Symposium on Service-Oriented System Engineering (SOSE). IEEE, 2017.

[Tsai 2018a] W. T. Tsai, Z. Zhao, C. Zhang, L. Yu, E. Deng "A Multi-Chain Model for CBDC." 2018 5th International Conference on Dependable Systems and Their Applications (DSA). IEEE, 2018.

[Tsai 2018b] W. T. Tsai, and L. Yu. "Lessons learned from developing permissioned blockchains." 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE, 2018.

[Tsai 2019] W. T. Tsai, and Christine Jinag, "Three Key Principles of Smart Contracts," Jan. 17, 2019. https://mp.weixin.qq.com/s/j5Ec2Jit69lsKOu1iexFUg

[Wang 2018] R. Wang, W. T. Tsai, J. He, C. Liu, and E. Deng, "A Distributed Digital Asset-Trading Platform Based on Permissioned Blockchains." International Conference on Smart Blockchain. Springer, Cham, 2018.

[Wood 2014] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum project yellow paper, vol. 151, pp. 1–32, 2014.

[Yu 2017a] L. Yu, W. T. Tsai, C. Hu, Baijie Li, J. Hu, and E. Deng "Modeling context-aware legal computing with bigraphs." 2017 IEEE Symposium on Service-Oriented System Engineering (SOSE). IEEE, 2017.

[Yu 2017b] L. Yu, W. T. Tsai, G. Li, Y. Yao, C. Hu, and E. Deng, "Smart-contract execution with concurrent block building." 2017 IEEE Symposium on Service-Oriented System Engineering (SOSE). IEEE, 2017.