SmaCoNat: Smart Contracts in Natural Language

Emanuel Regnath
emanuel.regnath@tum.de
Technical University of Munich, Germany

Sebastian Steinhorst
sebastian.steinhorst@tum.de
Technical University of Munich, Germany

***Abstract*—Smart contracts enable autonomous decentralized organizations (DAOs) in large, trustless and open trading networks by specifying conditions for automated transactions of cryptographically secured data. This data could represent cryptocurrencies but also sensor data or commands to Cyber-Physical Systems (CPS) connected to the Internet. To provide reliability, the contract code is enforced by consensus and the transactions it triggers are non-revertible, even if they were not intended by the programmer, which could lead to dangerous system behavior.**

In this paper, we conduct a survey over existing smart contract platforms and languages to determine requirements for the design of a safer contract language. Subsequently we propose concepts that enhance the understanding of code by limiting confusing language constructs, such as nesting, arbitrary naming of operations, and unreadable hash identifiers. This enables human reasoning about the contract semantics on a much higher abstraction layer, because a common understanding can be derived from the language specification itself.

We implement these concepts in a new domain specific language called *SmaCoNat* to illustrate the feasibility and show that our concepts are barely covered by existing languages but significantly enhance readability and safety without violating deterministic parsability.

***Index Terms*—Smart Contract, DSL, Blockchain, IoT, CPS**

I. Problem Statement

Smart contracts are scripts on a Blockchain that allow to securely automate multi-step trading of digital tokens in a decentralized network [1]. Beyond cryptocurrencies, these tokens can represent any piece of information in a complex decentralized process, such as items in a supply chain that are transfered between multiple parties, or a license key for an embedded firmware update that unlocks new capabilities.

Once a smart contract is deployed on the blockchain, its code cannot be altered and its behavior will be enforced by all participants in the network. While this enforcement offers security in the sense that parties can rely on the terms and conditions specified in the contract code, it also inherits a great risk, since any unintended execution path cannot be undone.

In 2016, an unknown attacker exploited an unintended behavior in one of the biggest smart contracts on the Ethereum platform called "The DAO" [2]. This contract was meant to securely automate the fund raising of a Startup but the attacker obtained $50 million worth of Ether currency. The smart contract was vulnerable to recursive calls using an overwritten default function, which allowed the attacker to withdraw money several times from *The DAO* account before it could update its balance correctly [3]. Some people argue that the exploit was part of the specification of the contract and thus legal.

With the support of the Technische Universität München – Institute for Advanced Study, funded by the German Excellence Initiative and the European Union Seventh Framework Programme under grant agreement n° 291763.

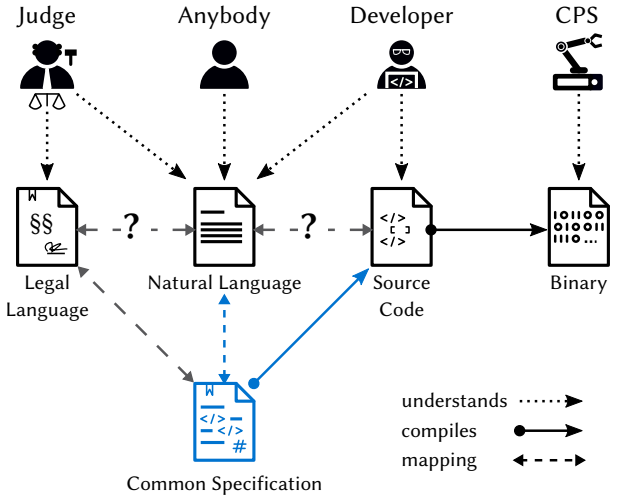

Figure 1: Our vision: A natural language specification that can be compiled to smart contract source code and is legally enforceable in court. To achieve this goal we need an unambiguous mapping between natural language and smart contract instructions.

However, since the attack involved 15% of all available Ether, the core developers decided to hard-fork the Blockchain, which means to create a completely new Blockchain that restarts from a block before the hack.

This example illustrates that the correctness of smart contracts according to their intended behavior is crucial for all involved parties. A common approach is to formally verify the program code against a specification that is considered to cover the intention. However, verification only shifts the problem to the specification because then the specification needs to be correct and complete by human reasoning. Thus, the behavior is eventually determined by some sort of code and we need to investigate how intention can be expressed clearly in code, such that also non-software developers can reason about it. Human intention is mostly specified in natural language, which is easy to understand for most humans but often highly ambiguous and subject to interpretation.

Although the small set of data types and operations of most programming languages is less ambiguous, programming code introduces an arbitrary mapping to natural language by allowing the developer to freely choose names for functions and data. We simply cannot trust a custom defined function `sum(a,b)` to correctly add `a` and `b` until we break it down to operations that are predefined by the language itself where we have a common understanding of their behavior.

Some domains, such as law, managed to create specifications that express conditions and intentions in a type of natural language that is less prone to misinterpretation *and* provides a common understanding that is continuously reinforced by the

decisions made in court.

In this paper, we want to investigate how natural language concepts can be used to create a smart contract specification language that is human readable, compilable to executable code, and legally enforceable, as it is illustrated in Figure 1.

A. Scope and Contributions

This paper addresses the challenges of smart contracts that arise from an ambiguous mapping between a contract's intention (natural language) and the program instructions that are finally executed. We then propose a methodology to create a high-level specification for program code that achieves a common understanding via natural language sentences and can be directly compiled to machine instructions. In particular, we

- conduct a short survey on smart contracts from which we identify language requirements (Section II),
- propose several concepts to create a mapping from natural language to program semantics (Section III),
- create *SmaCoNat*, a new domain specific language (DSL) that is tailored for a subset of the transaction logic we found in smart contracts (Section IV), and
- evaluate and compare *SmaCoNat* to existing languages.

II. State of the Art

In this section, we present the working principles of smart contracts, their current implementations and theoretical models, and derive a set of requirements.

A. Smart Contracts

The term smart contract was coined by Nick Szabo as "a set of promises, specified in digital form" [4] and a transaction protocol that enforces these promises. In general, smart contracts provide a marketplace of services concerning the "exchange and tracking of a digital asset" [1]. These digital marketplaces are proposed to be used for automation in many scenarios such as supply-chain tracking, energy trading in smart grids, property renting, or embedded firmware updates [1] and might be important for future decentralized CPS architectures.

In its modern implementation, which is shown in Figure 2, a smart contract is a program code that is stored and executed by a network of participating nodes. The nodes keep track of the ownership of all of all existing assets. The assets belong to accounts which in turn belong to the trading parties. A party could be either a human or a smart contract itself.

In its basic form, a smart contract specifies conditions on incoming transactions which will automatically trigger further transactions if those conditions are met. The accounts, which own the assets, are independent from the network nodes, since a network node is just the computer that runs the platform and not necessarily an entity that owns assets. However, in most scenarios nodes get paid for the computation in digital assets and thus need to hold an account as well.

The network nodes are responsible for validating and applying transactions as well as executing the instructions of a smart contract, which in turn could generate new transactions.

In most cases, the consensus about the ownership of assets is reached by using a Blockchain that keeps track of all transactions of assets that are ever made since the start of the network.

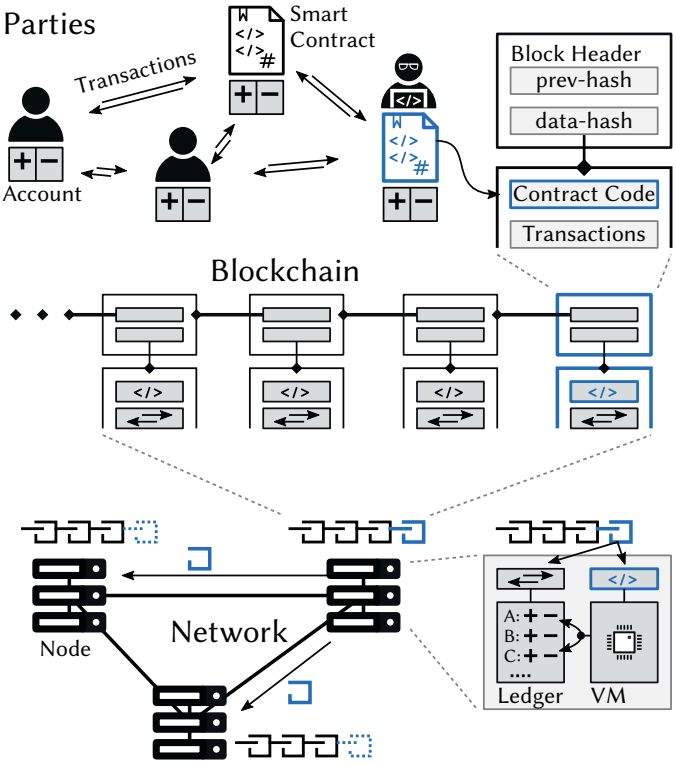

Figure 2: Architecture of a smart contract platform. Contract code and transactions between accounts are stored in a blockchain which is replicated and processed by network nodes.

1) Blockchain is a distributed data structure that stores data in an ordered chain of blocks and is shared and replicated by all nodes [1]. In cryptocurrencies, it functions as a ledger that stores transactions of assets and is therefore referred to as a Distributed Ledger Technology (DLT).

The blocks are divided into block-data and a corresponding block-header that stores the hash of the data. Each block also confirms and reinforces the data of its preceding block by including the hash of the previous block-header in its own header (see Figure 2). For this reason, any change to the Blockchain would result in changing hashes up to the latest block, which makes it impossible to change any accepted transaction in the system that was included in the Blockchain.

The current state of the system can be determined by applying all transactions on the initial state which is specified in the first block, called *Genesis Block*.

2) Transactions are broadcasted events that transfer the ownership of an asset. Transactions can be created manually by humans or automatically by smart contracts. In principle, a transaction is valid if it is cryptographically signed with the key-pair that belongs to the owner of the transfered asset. Nodes in the network will validate transactions and – if valid – include them in the blockchain, to confirm their validity. To keep track of the ownership of the transfered asset in a DLT, such as Blockchain, there are two common models:

First, the *account-based* model, where each node simply stores the balance of all assets that each account owns. Transactions simply reduce the amount of an asset on the sending account and increases the amount on the receiving account.

Second, the Unspent Transaction Outputs (UTXO) model,

where assets are stored on addresses that belong to cryptographic keys instead of accounts. Instead of adjusting balances for each address, a transaction fully "consumes" assets from a list of input addresses and reproduces them on a list of (new) output addresses.

The validity and the order in which transactions are added to the blockchain and applied is determined by an underlying consensus mechanism.

3) Consensus: The nodes in the network are responsible for keeping track of all transactions and executing the code of smart contracts. In order to keep a single, consistent state of the system, all nodes in the network need to reach consensus (agreement) on the validity and order of broadcasted transactions by voting on them.

While there are many different consensus protocols [5], currently the most common types for a Blockchain are Proof-of-Work (PoW) and Proof-of-Stake (PoS). The advantage of these two over traditional mechanisms, such as Practical Byzantine Fault Tolerance (PBFT), is that they enable a trustless, open network where everyone can join without the need to keep a list of permissioned nodes allowed to vote for a certain system state.

In PoW, the voting weight of any node is correlated to its processing power which is physically embodied and not duplicable for free like virtual identities. In order to vote, the nodes are required to solve a cryptographic puzzle by brute-forcing and they get paid for the effort when their vote is considered valid by consensus.

In PoS, the voting weight is correlated to the stake of a node within a crypto-currency system to avoid the energy waste of PoW. Nodes need to bind a certain amount of money to their vote and if it is considered correct they get paid back a higher amount. Otherwise, the money is lost. The consensus is achieved if at least $\frac{2}{3}$ of all voting power belongs to honest nodes.

B. Implementations

In order to identify implementation constraints, we will evaluate several platforms that support smart contracts. Our findings are summarized in Table I.

1) Bitcoin [6] started 2009 as the first pure digital cryptocurrency and established the *Blockchain* as tamper-proof DLT on which most other implementations are built upon. The Bitcoin Blockchain uses the UTXO model to keep track of the balances for each address. Bitcoin transactions also embed *Script*, a simple stack-based byte-code-language that specifies which conditions must be met (e.g. providing the correct signature) in order to spend the Bitcoins that were transfered by the corresponding transaction. The Script is a list of instructions that are linearly executed without backward jumps which leads to Turing-incomplete programs that will always terminate [7].

2) Ethereum is an account-based DLT with focus on decentralized general purpose computing. Accounts can optionally store contract code, which will be executed each time a triggering transaction is made to the corresponding account. This way, contract-controlled accounts can autonomously interact with each other, modeling complex multi-step processes. This feature, however, inherits the risk of creating an infinite loop between two accounts [8]. To solve this issue, contracts can only execute a certain amount of operations which is determined by the paid transaction fees of the triggering transaction.

Contracts are written as stack-based byte-code for the Ethereum Virtual Machine (EVM). The EVM is Turing-complete and can access the storage of an account which is an infinite byte array. For convenient contract creation, Ethereum offers Solidity [9], an object-oriented programming language based on JavaScript that can be compiled to EVM code. Ethereum also defined the ERC-20 Token Standard [10] which defines an unified API for tokens.

3) Neo Ecosystem is similar to Ethereum as it trades digital assets between parties using a smart contract framework that is executed on their own stack-based NeoVM [11]. However, the NeoVM allows only certain operations and NEO provides compilers from several well-known languages (e.g. C#, Java, Python) to NeoVM instructions [11].

4) NXT is a DLT that offers several *transaction templates* designed as basic communication mechanisms for the creation and trading of tokens [12]. These templates can be seen as fixed conditional contracts and the available features include, for example, asset trading, decentralized DNS, public polls, and encrypted messaging. A user can set up such a contract template by setting parameters in a web-interface and finally issue the contract as transaction to the NXT network. When another user transfers a certain token (e.g. money as NXT tokens) to the contract, it can automatically respond with another transaction when certain conditions are met.

5) Corda is a UTXO-based DLT for financial trading. It uses contract code that is linked to a legal prose to achieve automation and legal enforceability [13]. The smart contracts transfer *state-objects* between communicating parties. The state-objects can hold arbitrary business information and are processed by the contract code of the receiving party.

They identified *ownable* states as the fundamental building blocks for distributed ledgers from which they derive *fungible* assets. Fungible assets can – unlike unique tokens – be combined to represent a balance.

Contracts are executed as byte-code in a deterministic Java Virtual Machine (JVM) that allows only white-listed language constructs [14]. For example, contracts are limited to "pure"-functions that can only consume or append data on the state-object that was transacted to the contract function. Storing persistent state variables outside the state-object as well as using any random or time-based function is not possible.

The legal prose consists of a template text that is filled with parseable constant parameters and the hash of the legal prose is attached to the contract as reference in the case of a dispute.

6) Cardano is a UTXO-based, Proof-of-Stake DLT and uses its own contract language *Plutus*, which is inspired by Haskell [15]. Plutus therefore provides strongly typed, functional, general purpose programming [16]. However, Plutus does also allow arbitrary naming and thus does not provide any mechanism to link code to trading ontology.

7) Tezos is a self-amending, account-based smart contract platform that uses delegated PoS [17]. It focuses on formal verification of contract code.

Platform	Ledger, Consensus	OP Codes / Language	Features
Bitcoin	UTXO, PoW	Script† / Ivy	Linear execution conditions, no loops
Ethereum	Accounts, PoW→PoS	EVM / Solidity	General purpose computing
Neo	Accounts, D-BFT	NeoVM / C#, Java, ...	Many compilers for high-level languages
NXT	Accounts, PoS	Templates† / Website Forms	Just parameters, no coding
Corda	UTXO, Raft	JVM / Java, Kotlin	stateless functions, links legal prose
Cardano	UTXO, PoS	IELE / Plutus	functional programming
Tezos	Accounts, PoS	Michelson / Liquidity	formal verification

Table I: Different platforms that implement smart contracts.
†: language limited and *not* Turing-complete.

C. Theoretical Smart Contract Models

Despite specifying smart contracts with a programming language, we also found alternative approaches in literature, which we summarize shortly in the following. These approaches are mostly of theoretical nature and try to formalize concepts and requirements for smart contracts.

1) The Ricardian Contract is a model for digital traded assets in which assets are described as "contracts" between an issuer and a holder [18]. This method allows each participant in a trading system to issue own (competing) assets with its own set of trading rules, representing any type of value. These contracts consist of *legal text*, *parameters*, and a *signature chain* which all is digitally signed by the issuer.

By including the signing-key of the issuer in the contract itself, it contains its own PKI and only this top-level signing-key needs to be authenticated to belong to the real issuer in the beginning.

Any transaction in this system includes the hash of the contract that issued the transfered asset to secure the claims and prevent changes in the contract claims. The same contract should be readable by people and parsable by programs.

This contract concept is used by some systems, such as CommonAccord [19].

2) Smart Contract Templates which are described in [20], [21], extend the concepts of the Ricardian contracts and link legal agreements to executable code to achieve enforceability – either by law or by tamper-proof software execution.

Their contracts consist of two separate parts, the legal contract prose and the executable contract code. The legal prose is written in natural language which also includes parsable parameters. These parameters are used as configuration for a standardized, fixed executable code, whose behavior is only controlled by the provided parameters.

The parameters are key-value pairs that have an *identity* (key), a *type*, and a *value*. Parameters might be defined, assigned and referenced in different locations of the legal prose and could hold complex data structures. Using powerful parameters is necessary to use standardized code, which could be thoroughly tested and certified, in contrast to custom code that could lead to unintended behavior. [20]

The authors further sketch the idea that parameter values could also be expressions based on other parameter values or that a structured language could allow to directly write the expression into the legal prose [20]. They also expect that long-term research will lead to a language that can be compiled to executable code and is legally binding at the same time.

To enable this vision, the template system needs to satisfy several requirements, such as a common ontology that allows reasoning about the semantics of the contract, as well as a structured separation of large agreements into logical parts, such as definitions, obligations and schedules [21].

Overall, the presented template system uses a separation of code and prose. Increasing the parameter complexity for standardized code decreases the verification effort for instructions but increases the complexity for verifying that the ranges and combinations of parameter values are valid. At the end, a human still needs to understand what might happen, before the code is executed.

3) Ontology for Smart Contracts: The work in [22] suggests an ontology for smart contracts that uses *Agent*, *Commitment*, *Transaction*, and *Resource*. Smart contracts are defined as *Commitments* between *Agents*, which automatically execute *Transactions* of a certain *Resource*.

D. Discussion

We presented an overview on smart contracts, their current implementations, and ongoing research in alternative specification methods.

We found that most existing implementations either execute low-level byte code in a Turing-complete virtual machine (VM), or restrict contract capabilities to fixed templates that offer simple conditional execution of transactions. Since bytecode languages are difficult to write and read, some offer compilers from high-level programming languages.

The high diversity of the languages used to program smart contracts illustrates the problem that there is no suitable known language yet and the platforms often attempt to create such a language by their own.

Most implementations also identified the need to limit language constructs to achieve a deterministically terminating program which achieves some safety on the execution level. However, on the semantic level, there is almost no effort to provide safety by providing a common understanding for all involved parties. All considered programming languages offer infinite aliasing of operations and data structures.

All smart contract models use the business concept of a specific predefined currency and only allow additional tokens to be issued and traded. However, currencies and tokens could both be derived from the model of a generic asset that is traded between parties.

What is also missing is the concept of permissions. Accounts need to authenticate via a signature to transfer their assets but beyond that there are no restrictions for transactions of assets. However, we think that permissions could also be considered for the trading and issuing of assets.

The alternative approaches from literature separate the executed code from the legal prose and link these two together via parameters. This enables a natural description of the intention in the legal prose while the code could be more standardized. However, there is no way to determine whether the legal prose actually matches the contract code or how any dispute can be resolved in case the code does not completely behave the way

it is stated in the prose. Overall, we think that a separation of code and prose is just a temporary solution that introduces new types of problems.

E. Requirements

We use the results from our survey and discussion to identify requirements that are elementary for smart contract applications and constrain the design space for a smart contract language.

As we have seen from existing implementations, the execution of transactions including validation of signatures will be handled by the underlying DLT, such as Blockchain and PoW, and thus does not necessarily need to be expressed by the contract language. On the other hand, application specific semantics about currencies, sensor values and commands in the domain of IoT are probably too broad to be covered in a single language. Within the IoT, we therefore focus only on the trading aspect of CPSs that exchange data by treating this data as an asset that can be owned and transfered without considering its actual meaning. We see trading as the first and fundamental layer of behavior that a smart contract specifies and enforces. However, further layers could subsequently extend a contract's expressiveness to the application domain in the future.

a) Trading Ontology: Smart contracts are about trading digital assets with different properties and therefore the contract language should provide keywords and operations in natural language terms that are already used and understood in the real world domain of trading to ease human reasoning about the semantic.

b) Ownership Management: The type system of the language needs to be able to model owning parties and owned assets. All existing assets need to be globally identified and assigned to a specific owner. If assets can also be created by parties, the type system should also represent which assets were created by which party.

c) Trading Logic: Smart contracts need to express the logic for trading assets. In its basic form, it needs to specify conditions on a received transaction which will trigger other transactions. This logic could be formalized in general as

$$\begin{gathered}\text{When } A \text{ transfers } x \text{ with properties } P_x \text{ to } B,\\ \text{then } B \text{ transfers } y \text{ with properties } P_y \text{ to } C\end{gathered}$$

where A, B, C are accounts and x, y are assets. A and C could also refer to the same account, to express a kind of exchange contract. To express conditions on properties, the language also needs to express mathematical relations for the comparison of properties.

III. Our Approach

In this section, we illustrate concepts that can help to design a natural language-oriented specification for smart contracts that allow human reasoning on a high abstraction layer.

Our overall vision is a language that is human readable, safe to use, legally binding, and executable. While this goal seems very ambitious, we want to evaluate concepts that can help to approach it. We focus on two main problems that we think are crucial: readability and safety.

First, current programming languages are hard to read for humans because they are designed to be parsed by compilers and therefore enforce a syntax that contradicts many aspects of natural language. While natural language is context-sensitive and ambiguous, it provides a common understanding by all humans and is much easier to read.

Key-Value	Natural Sentence	Hybrid

`entity: {`
`  "type": "Account"`
`  "name": "Bob"`
`  "issuer": "Alice"`
`  "year": "2018"`
`  "fund": "42 BTC"`
`}`
	`Account "Bob"`
	`issued`
	`by "Alice"`
	`on "2018"`
	`owns "42 BTC".`
		`Account "Bob"`
		`issued by "Alice"`
		`with`
		`{`
		`  year: "2018",`
		`  fund: "42 BTC"`
		`}.`

Table II: Different approaches for specifying properties. A list of key-value pairs could also be specified in a natural sentence using prepositions.

We therefore propose that we should shift programming language syntax towards natural language sentences as long as they can be compiled deterministically to executable machine instructions. For example, giving names to variables instead of directly using memory addresses was a huge step to improve human readability and it did not restrict the ability to compile such a language to machine instructions.

Second, current programming languages are unsafe in the sense that it is easy to write code that expresses a behavior that is not intended. One reason is that only a few operations are defined by the language itself and that a programmer is allowed to create new functions with arbitrary names. We therefore propose that we can improve language safety by reducing the possibility to repetitively alias logic and data structures by custom names.

In the following, we explain our specific concepts in more detail.

A. Limit Custom Naming

One source of ambiguity is the possibility to choose own function names. While the sectioning of code into custom functions is fundamental to most programming languages in order to handle complexity, it also allows to alias operations with arbitrary names. Humans and parsers are required to resolve each function name until there are only predefined operations such as mathematical arithmetic.

We should limit aliasing, such that a human only needs to resolve a few names before reaching predefined operations that are built upon common understanding. Finding a suitable balance between the amount of predefined operations and the amount of allowed aliasing would be a task for future studies.

B. Limit Nesting

Another concept that is heavily used in programming languages but not in natural language sentences is nesting of statements. For example, `if`-statements are often nested in a programming language but in natural language we would rather define a list of conditions concatenated using *"and"* or *"or"*.

We should limit nesting of logical structures and should aim for a more sequential specification as it occurs in natural sentences.

C. Sectioning the Code Structure

Most documents group the text into sections. For example, within the first pages, special terms and acronyms are defined

and later used in the text. In most programming languages, there is no structure enforced, hence allowing the programmer to declare and define data types any time.

Consequently, we propose a strict separation of data declaration and operational statements. The entire contract code should be sectioned allowing only certain language constructs in each section.

D. Predefined Type System

Most programming languages use a type system that provides only some base types of data, such as Integers, Floats, and Strings and then allow the programmer to define new types derived from them.

Weakly typed languages, such as Python, are most ambiguous because variables can change the type of data they represent by implicit type conversions. Conventional typed languages, such as C, assign each variable an explicit type but variables of the same type may be mixed even they represent different quantities. Strongly typed languages such as Ada, allow to derive distinct types from the same base type, which are incompatible to each other and may only be mixed by explicit type conversion.

For natural language, we can observe that many types are already implicitly defined. For example, a "Temperature" is completely incompatible to "Velocity", even though both could be represented by real numbers. However, synonyms, such as "Velocity" and "Speed", lead to confusion about compatibility and should be avoided.

For smart contracts, we therefore suggest that each data type or data-structure should be predefined. This is possible because we only focus on the trading logic between CPSs and do not try to cover all possible application scenarios of data structures. Programmers should be only allowed to assign values to these predefined types, which could then be evaluated on a higher application layer. This way, the value of an asset or token could encode a complex data structure as string using, e.g., the JSON format, but this string value would remain meaningless for the semantics of the contract.

E. Natural Language Syntax

Programming languages define special keywords and symbols that are often not or only partially related to a natural language meaning. A statement in a programming language would be easier to understand if it reads like a natural sentence. To achieve this, all keywords and all identifiers in a smart contract should be meaningful words. These keywords should provide one context-insensitive meaning and should be easily distinguishable from each other. Table II illustrates how prepositions could be used to specify properties of a data structure in a natural sentence.

From analyzing the smart contract platforms and theoretical frameworks, we identified the following list of terms that could provide a trading ontology by answering the questions about *Who*, *What* and *How*:

- Who: Entity, Party, Account, Agent, User, Actor
- What: Data, Object, State, Message, Asset, Item, Token, Quantity, Currency, Value
- How: Transaction, Event, Action, Transition

Another step towards a natural language syntax is a reduced use of symbols. For example, in `C`, the symbol `&` is used as a boolean operation, for accessing a memory address and for declaring a reference. These different use cases make it hard to understand `C` code, especially for beginners, since there is almost no correlation to the natural meaning of the `&` symbol. We should also limit the need for parentheses or other delimiters to group several statements or expressions, because nesting of delimiters is a typical source of confusion. For example, instead of using `{` and `}` to mark the body of an `if`-statement, we should use `then` and `end if`, as it is already done by some languages.

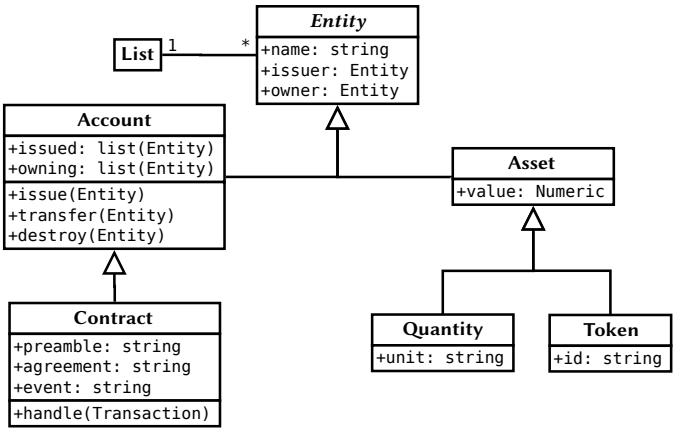

Figure 3: UML diagram of our proposed data structures.

F. Human-readable Global Identifiers

Since smart contracts allow everyone to globally register accounts and globally trade assets, we need global identifiers. When it comes to globally identifying data, e.g. specific transactions, cryptographic hashes are the common choice for smart contract platforms because they provide enough entropy to be unique. However, hashes are not human readable and could be easily confused. We should aim for natural language identifiers instead. A successful implementation of this idea are domain names, which are more readable than IP addresses.

We therefore suggest to use a scope system similar to URLs, but based on the issuer of an account. An account would then be identified by its name and the name of its issuer, which could result in an identifier such as `licensekey.alice.company` to identify a license key issued by the account "Alice" which in turn was issued by a globally known "Company". This would allow any account to issue its own version of an asset called `licensekey` that could be traded.

IV. SmaCoNat: Our Natural DSL

In this section, we use the previously described methods to propose a new domain specific language, we call *SmaCoNat* (*Sma*rt *Co*ntracts *Na*tural), suitable to express smart contract behavior in a natural language syntax. We do not aim for a full-featured language but rather illustrate how to implement our concepts for a small set of types and operations. We implemented SmaCoNat with *Xtext* [23], a framework for developing DSLs that is part of the Eclipse Modeling Framework. All code examples in the remainder of this section, on how we implemented the language are given in simplified EBNF syntax instead of full Xtext syntax.

A. Type System and Trading Ontology

As discussed before, all data types will be predefined. We model a smart contract by using the standard primitive types, such as Integer and Strings, and a tree-structured hierarchy of a few composite types, which are shown in Figure 3.

Since smart contracts automate trading, we use a minimal ontology for our data model that captures the very nature of trading: Ownership. Therefore, the common abstract class is an *Entity*, which represents everything that can be created and possessed.

From an *Entity*, we derive the concrete types *Accounts* and *Assets*. *Accounts* are the actors/agents in the system that transfer *Assets*. *Assets* are any information that can be traded such as currencies, tokens, or sensor data. *Assets* are *issued* or *revoked* by an *Account*, and can be *transfered* from one *Account* to another.

B. Enforced Structure

In contrast to languages such as `C`, where a valid program is simply an unstructured list of statements, we enforce a certain structure on the first level of the code. Thus, we first define the whole contract code as an ordered sequence of five distinct rules:

```
Contract = Heading, AccountSection, AssetSection,
     AgreementSection, EventSection;
```

The *Heading* states the contract language and the version of the language. After the heading, all involved accounts and assets must be declared. The agreement section specifies the behavior that will be executed once the contract is signed by all involved accounts. The events specify the behavior of the signed contract when an asset is transfered to the contract. Agreements and events are only allowed to refer to previously declared accounts and assets.

C. Global Identifiers

Identifiers for entities must explicitly name the type followed by the Account names of the chain of all involved issuers until a known Account alias is reached. For example, an account identifier is defined as

```
AccountId = 'Account', NAME, ('by' NAME)*, 'by', AccountAlias;
```

where `AccountAlias` refers to a list of globally known special accounts or a previously defined account alias and `NAME` is a terminal rule that matches strings enclosed in single quotation marks. We defined the three special account aliases `Self`, `Genesis`, `Anyone` and one special asset alias `Input`.

`Self` matches the Account belonging to the contract. `Genesis` is the Account that issued and owns all entities in the initial state of the distributed ledger and has no issuer/owner itself. `Anyone` matches any Account and has no issuer. The asset `Input` refers to the asset that was sent to the contract.

a) Single Aliasing To avoid repetition of long and unreadable global identifiers, it is allowed to alias accounts and assets during their declaration. For example, the `AccountSection` rule is defined as

```
AccountSection =
   '§ Involved Accounts:',
   (AccountId, ('alias', NAME)?, '.')*
;
```

D. Logic

1) Operations: A contract may perform basic arithmetic operations on the primitive types. For asset types we define only three fundamental operations:

```
ASSETOP = 'issue' | 'transfer' | 'revoke';
```

2) Conditions: The contract may also contain non-nested conditional statements on boolean expressions. Boolean expressions consist of the equality relation (`equal to`) for all types and the additional relations `smaller than` and `larger than` for numeric primitive types. All relations may be negated by perpending the keyword `not`.

V. Evaluation

In general, it is difficult to evaluate a programming language for its safety and expressiveness. In this section we give an example for a valid SmaCoNat contract showing its feasibility to express a typical contract behavior and finally compare other languages regarding our language concepts. The results are summarized in Table III.

A. SmaCoNat Sensor Example

The following contract example, written in SmaCoNat, specifies the behavior of a CPS that manages 42 parking lots by selling parking tickets and controlling the parking barrier. In this scenario involved are the controller and two barriers from a globally known company *AComp* as well as any vehicle approaching the barriers. One ticket costs 0.3 units of the globally known currency *TheCoin* which was issued by the Genesis block.

```
1  Contract in SmaCoNat version 0.1.
2
3  § Involved Accounts:
4  Account 'BarrierIn' by 'AComp' by Genesis alias 'BarrierIn'.
5  Account 'BarrierOut' by 'AComp' by Genesis alias 'BarrierOut'.
6
7  § Involved Assets:
8  Asset 'TheCoin' by Genesis alias 'TheCoin'.
9  Asset 'ParkTicket' by Self alias 'Ticket'.
10 Asset 'OpenBarrier' by Self alias 'Open'.
11
12 § Agreement:
13 Self issues 'Ticket' with value 42.
14 Self issues 'Open' with value 1.
15
16 § Input Event:
17 if Input is equal to 'TheCoin' from Anyone
18 and if value of Input is equal to 0.3
19 then
20    Self transfers 'Ticket' with value 1 to owner of Input.
21    Self transfers 'Open' with value 1 to 'BarrierIn'.
22    Self issues 'Open' with value 1.
23 endif
24
25 if Input is equal to 'Ticket' from Anyone then
26    Self transfers 'Open' with value 1 to 'BarrierOut'.
27    Self issues 'Open' with value 1.
28 endif
```

While this example is very simplified, it illustrates how a system functionality can be mapped to a smart contract. Furthermore, we do not need to perform a lot of checks, such as checking the remaining tickets or validity of tickets because this will be handled on the lower transaction layer by the network.

	SmaCoNat	Solidity	Plutus	Liquidity	Rholang
Structure	*section*	function	function	*section*	function
Typing	*predefined*	strong	strong	strong	behavioral
Aliasing	*single*	infinite	infinite	infinite	infinite
Ontology	*trading*	general/ *trading*	general	general/ *trading*	general
Global IDs	*names*	hash	hash	hash	hash
Special Symbols	*few*	some	some	many	many

Table III: Evaluation and comparison of our DSL against other contract languages.

B. Comparison

1) Enforced Structure: Enforcing a sectioned code structure is a step towards code safety. In contrast, most languages such as Solidity just group statements to functions but allow any structure within a group. Its grammar [24] on the top level is defined as

```
SourceUnit = (PragmaDirective | ImportDirective |
     ContractDefinition)*
```

and within the body of `ContractDefinition` any order of statements is allowed. Only Liquidity enforces a code structure on the top level such that type declarations are only allowed before the entry point. In contrast, SmaCoNat strictly enforces a structure that separates different language aspects, making it easier to analyze the code.

2) Type System: Almost all languages use a static and strong type system and allow the programmer to create own types. This does not only introduce aliasing of data instances but also aliasing of data structures. Rholang uses behavioral types, which are worse in the sense that a programmer is allowed to specify custom behavior for the types making types another source of custom named behavior. SmaCoNat only uses predefined types which provide a common understanding.

3) Expressiveness: Some concepts that make SmaCoNat safer and more readable also pose limitations to the expressiveness of the language. For example, we did not consider loops, which could be enabled to some extend by allowing iterations over lists of entities. While the other languages are considered Turing-complete, most smart contracts do not require loops and Turing-completeness [8]. Overall, we believe that SmaCoNat can already express a wide range of behaviors despite the restrictions we put on the language.

4) Trading Ontology and Identifiers: Solidity predefines some operations on their `address`-type, such as `balance` and `transfer` that are specific for trading.

Liquidity also defines trading-specific functions such as `Account.create()` and `Current.balance()`. All other languages only use a general-purpose computing ontology which makes it impossible to reason about the semantic on a higher abstraction layer. For identifying transactions and accounts, it seems that all considered languages use hash digests that can be assigned to variables with custom names, which makes them more readable but also introduces the aforementioned issues. All considered languages use symbols instead of words for all delimiters and partially for operations. While the general usage in Solidity and Plutus can be considered moderate, Liquidity and Rholang make heavy use of symbols to encode semantics. Rholang even overloads symbols with different meanings depending on the context. SmaCoNat allows symbols only for arithmetic operations and punctuation of natural language such as the period `'.'` to mark the end of a statement.

VI. Conclusion

Smart contracts are promising for secure automation in IoT environments. However, for broad acceptance, we need a readable and safe contract specification that can be directly compiled to executable instructions. Existing implementations lack a well-defined mapping to natural language, prohibiting human reasoning on higher abstraction layers.

Therefore, we derived several language design concepts that can be used to narrow the gap between conventional source code and natural language descriptions to approach a unified contract language. We implemented a DSL called *SmaCoNat* by predefining a small set of operations and data types that allow to directly express the trading logic with predefined operations and prevents custom naming of identifiers. In contrast to existing smart contract languages, *SmaCoNat* enforces a clear code structure, limits aliasing, and builds purely on natural language identifiers, hence enabling a common understanding of code semantics on higher abstraction layers.

References

[1] K. Christidis and M. Devetsikiotis, "Blockchains and Smart Contracts for the Internet of Things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.
[2] K. Finley, "A $50 million hack just showed that the DAO was all too human," *Wired Business*, 6 2016.
[3] P. Daian, "Analysis of the DAO exploit," *Hacking, Distributed*, 6 2016. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit
[4] N. Szabo, "Smart Contracts: Building Blocks for Digital Markets," 1996.
[5] C. Cachin and M. Vukolic, "Blockchain Consensus Protocols in the Wild (Keynote Talk)," in *31st International Symposium on Distributed Computing (DISC 2017)*, vol. 91, 2017, pp. 1 – 16.
[6] S. Nakomoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. https://www.bitcoin.org/bitcoin.pdf
[7] P. L. Seijas, S. J. Thompson, and D. McAdams, "Scripting smart contracts for distributed ledger technology." *IACR Cryptology ePrint Archive*, vol. 2016, p. 1156, 2016.
[8] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *GitHub*, Jan. 2015.
[9] Ethereum Foundation, *The Solidity Contract-Oriented Programming Language*. https://github.com/ethereum/solidity
[10] F. Vogelsteller and V. Buterin, "Erc-20 token standard," 11 2015. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
[11] NEO, "NEO white paper," http://docs.neo.org/en-us/index.html.
[12] NXT community, "Nxt whitepaper," 7 2014, revision 4,.
[13] R. G. Brown *et. al.*, "Corda: An introduction," 2016, corda.net.
[14] M. Hearn, "Corda: A distributed ledger," 11 2016.
[15] Cardano, "Why we are building cardano," https://whycardano.com.
[16] ——, "Cardano settlement layer documentation," cardanodocs.com.
[17] L. Goodman, "Tezos — a self-amending crypto-ledger," 9 2014.
[18] I. Grigg, "The ricardian contract," in *Proceedings of the First International Workshop on Electronic Contracting*. IEEE, 7 2004.
[19] J. Hazard and H. Haapio, "Wise contracts: Smart contracts that work for people and machines," in *Proceedings of the 20th International Legal Informatics Symposium*, 2 2017.
[20] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: Foundations, design landscape and research directions," *arXiv preprint arXiv:1608.00771*, 8 2016. http://arxiv.org/abs/1608.00771
[21] ——, "Smart contract templates: essential requirements and design options," *arXiv preprint arXiv:1612.04496*, 12 2016.
[22] J. de Kruijff and H. Weigand, "Ontologies for commitment-based smart contracts," in *OTM Confederated International Conferences*. Springer, 2017, pp. 383–398.
[23] S. Efftinge and M. Spoenemann, "Xtext – language engineering made easy!" https://www.eclipse.org/Xtext, accessed 2018-04-03.
[24] Ethereum Foundation, *The Solidity Contract-Oriented Programming Language*. http://solidity.readthedocs.io/en/develop/miscellaneous.html#language-grammar