Domain Specific Language for Smart Contract Development

Maximilian Wöhrer, Uwe Zdun
Faculty of Computer Science, Research Group Software Architecture
University of Vienna, Vienna, Austria
{maximilian.woehrer,uwe.zdun}@univie.ac.at

***Abstract*—The notion to digitally articulate, execute, and enforce agreements with smart contracts has become a feasible reality today. Smart contracts have the potential to vastly improve the efficiency and security of traditional contracts through their self-executing autonomy. To realize smart contracts several blockchain-based ecosystems exist. Today a prominent representative is Ethereum. Its programming language Solidity is used to capture and express contractual clauses in the form of code. However, due to the conceptual discrepancy between contractual clauses and corresponding code, it is hard for domain stakeholders to easily understand contracts, and for developers to write code efficiently without errors. Our research addresses these issues by the design and study of a domain-specific smart contract language based on higher level of abstraction that can be automatically transformed to an implementation. In particular, we propose a clause grammar close to natural language, helpful coding abstractions, and the automatic integration of commonly occurring design patterns during code generation. Through these measures, our approach can reduce the design complexity leading to an increased comprehensibility and reduced error susceptibility. Several implementations of exemplary smart contract scenarios, mostly taken from the Solidity documentation, are used to demonstrate the applicability of our approach.**

I. Introduction

A contract is a "promise or a set of promises, for the breach of which the law gives a remedy, or the performance of which the law in some way recognizes as a duty" [1]. Contracts are common in almost every facet of the business world. Like in many other areas, the trend towards digitization has also taken hold in this field and led to the concept of smart contracts [2], [3]. Smart contracts are a means to digitally facilitate, verify, and enforce the negotiation or execution of contracts and "represent a new era of contracting" [4]. This evolution is grounded on several technological advancements and transformations. Blockchain technology, with its underlying consensus mechanism (implemented through different protocols), allows various parties to reach agreements without requiring any trusted participants among them. This feature paved the way for a decentralized exchange of digital assets (cryptocurrencies), and the subsequent inclusion of general scripting languages enabled the evolution towards distributed computing platforms. Both features, the build-in exchange of digital assets (as a means of payment) and the dispersed code execution (supporting distributed applications), are the prerequisites for an ecosystem that makes the notion of smart contracts feasible. Today's predominant ecosystem in this regard is Ethereum [5], a blockchain based distributed computing platform, that allows to formulate smart contracts in the platform's leading programming language Solidity.

The formalization of contracts in a machine-readable and executable form is a challenging task. Mapping the broad articulation space of contracts written in natural language to a conclusive and unambiguous digital representation requires a formalization approach to deduce a proper digital manifestation. In the context of Ethereum, this means translating contract statements from natural (legalese) language into equivalent Solidity code. There is therefore a high likelihood of translation loss. To make matters worse, the blockchain runtime environment and missing high-level abstractions complicate writing correct and secure smart contracts for Ethereum and other blockchain technologies even further.

Our work investigates how productivity can be increased in smart contract development and how to address the aforementioned issues with a domain specific language for smart contract formulation called Contract Modeling Language (CML). A domain specific language (DSL) is a programming language of limited expressiveness focused on a particular domain [6]. When used properly DSLs can improve productivity by simplifying complex code, promoting communication between domain stakeholders, and eliminating development bottlenecks.

The objective of CML is to investigate how unstructured legal contracts can be uniformly modeled and specified (covering a variety of common contract situations) in order to improve their interpretation and the automatic generation of smart contract implementations. In particular we focus on a declarative and imperative formalization, since we are interested in the conceptual representation of contracts in a programming language. In this context our work seeks to address the following research questions (RQs):

- RQ1 How and in how far is it possible to bring the abstraction level of smart contracts closer to the contract domain?
- RQ2 Can higher abstraction levels in combination with code generation (considering platform-specific programming idioms) reduce the risk of smart contract errors?

The paper is organised in the following way: First, we provide a short background on contracts and their building blocks in Section II and our research methodology in Section III. Then, we present our domain specific language in Section IV, before we illustrate its practicality in Section V, and evaluate

978-1-7281-6680-3/20/$31.00 © 2020 IEEE

and discuss our findings in Sections VI and VII respectively. Finally, we compare to related work in Section VIII, and then draw conclusions in Section IX.

II. Background

Although contracts cover a wide range of subject areas, most will share many common features, as exemplified by the sale of goods contract excerpt in Fig. 1. Regarding their form, contracts usually build on structuring techniques such as sectional delimitation and paragraph division. Typically contracts are split into articles, sections, subsections and paragraphs which are numbered to support referencing and grouping of particular provisions. This organization can be interpreted as macro and micro structure that eases the legibility and comprehensibility of contracts.

Regardless of the subject matter and contract type, contracts share basic building blocks that serve the same function in all contracts [7]. First, there are (A) definitions which isolate and specify important key terms and concepts that are usually repeated in the agreement. Definitions make the contract more consistent and easier to read as unnecessary repetitions are avoided (e.g., §1.). Second, there are (B) covenants, which are promises made by a party to undertake or refrain from certain actions in the future (e.g., §3. to §5. and §7.). These are the most important provisions of any contract and constitute the obligations of the involved parties to certain performances. Covenants are typically reciprocal, e.g., one party is obligated to pay, while the other is obligated to perform. Covenants are often scattered throughout the contract and are organized by subject matter (e.g., payment obligations, performance obligations). A party's failure to satisfy covenants typically entitles the other party to certain remedies. Third, there are (C) representations and warranties, which are statements of fact made by the parties to each other as of a particular point in time (e.g., §9., §10.). These statements assert the truth about assumptions that are important for the decision to enter the contract and are implicitly coupled with indemnification obligations when untruthful. Fourth, there are (D) conditions, which specify certain requirements that must exist so that a party is obligated to perform under the contract (e.g., §8.). Theses conditions can be classified by type into conditions precedent and conditions subsequent. Conditions precedent specify the events that must occur to start one's duties to perform under the contract, likewise conditions subsequent specify the events that must occur to end one's duties to perform under the contract. Table I gives an overview of essential building blocks regarded important for contract construction.

CONTRACT FOR THE SALE OF GOODS

Paragraph 1. [_], hereinafter referred to as Seller, and [_], hereinafter referred to as Buyer, hereby agree on this [_] day of [_], in the year [_], to the following terms.

A. Identities of the Parties

Paragraph 2. Seller, whose business address is [_], in the city of [_], state of [_], is in the business of [_]. Buyer, whose business address is [_], in the city of [_], state of [_], is in the business of [_].

B. Description of the Goods

Paragraph 3. Seller agrees to transfer and deliver to Buyer, on or before 2019-09-01, the following goods: 1 x production line machinery at the price of $7,500.

C. Buyer's Rights and Obligations

Paragraph 4. Buyer agrees to accept the goods and pay for them according to the terms further set out below.

Paragraph 5. Buyer agrees to pay for the goods half upon receipt, with the remainder due within 30 days of delivery. If Buyer fails to pay second half within 30 days, an additional fine of 10% has to be paid within 14 days.

Paragraph 6. Goods are deemed received by Buyer upon delivery to Buyer's address as set forth above.

Paragraph 7. Buyer has the right to examine the goods upon receipt and has 14 days in which to notify seller of any claim for damages based on the condition, grade, or quality of the goods.

Paragraph 8. The Buyer's obligation to complete the purchase of the goods is subject to the Buyer obtaining a financing commitment of at least $5,000.

D. Seller's Obligations

Paragraph 9. Until received by Buyer, all risk of loss to the above-described goods is borne by Seller.

Paragraph 10. Seller warrants that the goods are free from any and all security interests, liens, and encumbrances.

Fig. 1. An exemplary contract for the sale of goods.

TABLE I
BASIC CONTRACTUAL BUILDING BLOCKS

	Covenant	Representation	Warranty	Condition
is a	promise	statement	statement or promise	statement
of	action or inaction	fact	fact or condition	condition
applies to	future	past or present	present and future	future
purpose	define activities that will (not) be carried out	make assurances to induce parties to enter contract	assure that facts and conditions are/will be true	define conditions affecting the party's contractual duty

III. Research Methodology

Our approach to design a DSL is guided by the design science methodology where "knowledge and understanding of a problem domain and its solution are achieved in the building and application of the designed artifact" [8]. In particular, we employed an approach described by Wieringa [9], where the design process iterates multiple times over two activities: first designing an artifact that improves something for stakeholders (design cycle) and subsequently empirically investigating the performance of that artifact in its context (empirical cycle). Our focus was on the design cycles, where an improvement problem is investigated, alternative treatment designs are generated and validated, a design is selected and implemented, and experience with the implementation is evaluated. In our context, this meant to investigate smart contract implementation issues, to come up with possible abstract language constructs, implement these constructs in a DSL development framework, and subsequently evaluate and asses the suitability of the implementation. For the empirical cycles, we performed an analysis of multiple scenario cases to evaluate the improvements in the design cycles (Section VI).

IV. Contract Modeling Language (CML)

CML is a high-level DSL using object-oriented abstractions for implementing smart contracts. It is designed with several intentions in mind. First, it should allow for the specification of common relevant contractual elements. Second, it should be easy to read and understand through a clause grammar close to natural language that resembles real-world contracts. Third, it should improve productivity and simplify complex code. Fourth, the defined contract logic should serve as basis for

code generation, backed possibly by a variety of distributed ledger technologies. Regarding the last intention, for proof-of-concept, we focus on the generation of Solidity code, being the predominant language for smart contracts today.

The CML language is developed in Xtext [10], a framework for the development of programming languages and DSLs. For reproducibility of our research, the CML language implementation source code is available on GitHub [11]. Further a CML web editor [12] for demonstration purposes exists.

A. Language Characteristics

The basic structure of a CML contract is similar to a class in object orientation. It consists of state variables and functions (actions), which read and modify these. In addition, a contract contains clauses, which mimic and capture covenants in a standardized way, close to natural language syntax. These indicate the context under which the actions are to be called, meaning they combine different aspects that influence action execution. In its most simplistic form a clause specifies the obligation or permission of a party to execute a specific action.

B. Type System

The simplest of types are primitive types which describe the various kinds of atomic values allowed in CML. These include *Boolean*, *String*, *Integer*, *Real*, *DateTime*, and *Duration*. The last two types represent the basic temporal concepts of absolute and relative time, needed to express temporal constraints and relationships typically encountered in contracts. Regarding temporal constrains and their verification in the context of contracts we refer interested readers to [13]. Beyond the aforementioned primitives, CML includes predefined and easily extensible structural composite types that are derived from literature on smart contract ontologies [14], [15], to embody common contract-specific concepts. These include *Party*, *Asset*, *Transaction*, and *Event*. *Party* denotes an individual or organization with an unique identifier that participates in a contract. *Asset* describes a resource (long-lived identifiable item) with a certain economic value. *Transaction* is used to describe a message that is submitted by a party along contract interaction. *Event* characterizes anything that happens, being either important or unusual. In addition, a few special variables (*caller*, *anyone*, *now*, *contractStart*, *contractEnd*) are defined which are always present and often needed during contract definition.

C. Clause Structure

CML introduces clauses as syntactical elements which are based on the covenants discussed in Section II. Covenants are most relevant for smart contracts, since they enclose the expected actions to be performed. In view of the dynamics of the natural language, in which they are represented, their composition cannot be precisely defined. However, there are structural components that can be singled out. Most clauses consist of (at least) three parts: an actor, an action, and a modality for that action. A very basic covenant reflecting these components is: “The buyer must pay.” Moreover, other commonly occurring components can be extracted, which specify the context of a covenant more precisely, such as trigger events, conditions, and involved objects. Trigger events stipulate under which circumstances a clause must be taken into account and refer to either internal or external events. Internal events can be controlled by the contract parties (e.g., satisfied action, fulfilled clause), whereas external events cannot be controlled by the parties themselves (e.g., price feed). Clause conditions define time and state restrictions that must be subsequently met after a trigger event. Involved objects specify the people, places, things receiving an action or having an action done to them. Fig. 2 shows a representation of fundamental covenant clause components discussed in this paragraph.

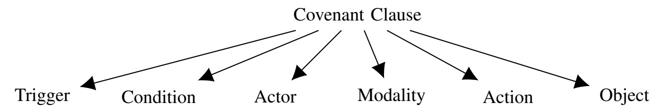

Fig. 2. Conceptual breakdown of covenant clause components.

Based on the above insights we propose a clause syntax, illustrated in Fig. 3, for the transformation of covenants. Each clause has an unique identifier for referencing and must contain at least an actor, an action, and the deontic modality of this action (i.e., “may” or “must”). Optional elements include temporal or general constraints. Temporal constrains are indicated by the keyword “due” followed by a temporal precedence statement (i.e., “after” or “before”) and a trigger expression. The trigger expression refers to an absolute time or a construct from which an absolute time can be deduced. This includes the performance of a clause, the execution of an action, or the occurrence of an external event. Additionally, the “due” statement can be enriched by a duration statement (“within”) to further specify the considered time-frame, as well as a repeat statement (“every”) to model the recurring nature of a covenant. General constraints can be defined after the keyword “given” by multiple linked conditions that evaluate to true or false. These conditions usually refer to the contract state, conditions regarding the transaction input are handled within the functions. It is worth noting that the deontic “must” requires the specification of a terminating temporal constraint (declared by “within”) to evaluate the fulfillment of a covenant, since without it, a covenant can always be met in the future.

clause ID
    [**due** [**within** RT] [**every** RT **from** AT **to** AT] (**after**|**before**) TRIGGER]
    [**given** CONDITION]
    party ACTOR
    (**may**|**must**) ACTION {(**and**|**or**|**xor**) ACTION}

Trigger: AT | ClauseTrigger | EventTrigger | ActionTrigger
ClauseTrigger: **clause** ID (**fulfilled**|**failed**)
ActionTrigger: ACTOR **did** ACTION
EventTrigger: **event** ID

RT...Relative Time, AT...Absolute Time

Fig. 3. Structure of a CML clause declaration.

D. CML by Example: Simple Open Auction

The application of clause constructs, predefined types, and type operations is shown by example. Listing 1 contains a CML contract specification for a simple auction in which anyone can bid during a bidding period. If the highest bid is raised, the previously highest bidder gets her bid back. After the end of the bidding period, the beneficiary can withdraw the highest bid.

```
namespace cml.examples

import cml.generator.annotation.solidity.*

@PullPayment
contract SimpleAuction
   Integer highestBid
   Party currentLeader
   Party beneficiary
   Duration biddingTime

   clause Bid
      due within biddingTime after contractStart
      party anyone
      may bid

   clause AuctionEnd
      due after contractStart.addDuration(biddingTime)
      party beneficiary
      may endAuction

   action init(Duration _biddingTime, Party _beneficiary)
      biddingTime = _biddingTime
      beneficiary = _beneficiary

   action bid(TokenTransaction t)
      ensure(t.amount > highestBid, "There already is a higher bid.")
      caller.deposit(t.amount)
      if (highestBid != 0)
         transfer(currentLeader, highestBid)
      currentLeader = caller
      highestBid = t.amount

   action endAuction()
      transfer(beneficiary, highestBid)
```

Listing 1. CML contract for a simple open auction.

V. Solidity Code Generation

The concrete syntax (grammar) of the CML is defined in Xtext, which generates the language infrastructure and derives a corresponding meta-model. Once a CML text input file is processed, the parser creates an in-memory instance of that meta-model, called abstract syntax tree (AST). This representation is then traversed by the generator, which is written in Xtend, to produce Solidity code that further relies on static and dynamically created support libraries. These libraries either contain the implementation of declared CML type operations, or relate to libraries for secure smart contract development. Fig. 4 illustrates this process.

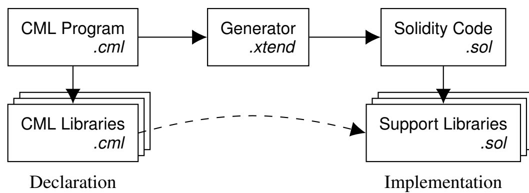

Fig. 4. The CML code generation process.

A. CML to Solidity Mapping

Having specified essential domain-specific constructs in CML through predefined types, we propose a mapping that allows an automated generation of Solidity contracts. The conceptual equivalent of CML domain model definitions (*Party*, *Asset*, *Transaction*) is Solidity's struct, in which the entire hierarchy of a given type is assembled. CML functions and enumerations are mapped to their respective conceptual equivalents in Solidity. Events that are contained in CML reflect external events and are mapped to functions in Solidity, since outside information can only enter the Solidity platform through interaction (passing data to a method). Regarding the constraints of clause constructs, these are reflected in a single function modifier that is added to clause functions and contains declarative checks that preempt improper execution. Table II summarizes the proposed CML-Solidity mapping.

To illustrate the mapping process, we are transferring the simple auction CML contract from Listing 1 to Solidity code presented in Listing 2. Beyond the generation of functional contract code, supportive code is generated to evaluate the execution context, which is required to apply any restrictions (e.g. time, caller, state) defined in the clause statements.

```
pragma solidity >=0.4.22 <0.7.0;
pragma experimental ABIEncoderV2;
import "./lib/cml/ConditionalContract.sol";
import "./lib/cml/DateTime.sol";
  ...
contract SimpleAuction is ConditionalContract, PullPayment {

  struct Party {
   address payable id;
  }

  uint highestBid;
  Party highestBidder;
  Party beneficiary;
  uint biddingTime;
  uint _contractStart;

  constructor(uint _biddingTime, Party memory _beneficiary) public {
   biddingTime = _biddingTime;
   beneficiary = _beneficiary;
   _contractStart = now;
  }

  function bid() public payable checkAllowed("Bid") {
   require(msg.value > highestBid, "There already is a higher bid.");
   if (highestBid != 0) {
    _asyncTransfer(highestBidder.id , highestBid);
   }
   highestBidder = Party(msg.sender);
   highestBid = msg.value;
  }

  function endAuction() public checkAllowed("AuctionEnd") {
   _asyncTransfer(beneficiary.id , highestBid);
  }
  ...
  function clauseAllowed(bytes32 _clauseId) internal returns (bool) {
   if (_clauseId == "Bid") {
    require(onlyAfter(_contractStart, biddingTime, true), "Function not called
          within expected timeframe");
    return true;
   }
   if (_clauseId == "AuctionEnd") {
    require(onlyBy(beneficiary.id), "Caller not authorized");
    require(onlyAfter(DateTime.addDuration(_contractStart, biddingTime), 0, false
          ), "Function called too early");
    return true;
   }
   return false;
  }

  function clauseFulfilledTime(bytes32 _clauseId) internal returns (uint) {
   uint max = 0;
   if (_clauseId == "Bid" && (callSuccess(this.bid.selector))) {
    if (max < callTime(this.bid.selector)) {
     max = callTime(this.bid.selector);
    }
    return max;
   }
   if (_clauseId == "AuctionEnd" && (callSuccess(this.endAuction.selector))) {
    if (max < callTime(this.endAuction.selector)) {
     max = callTime(this.endAuction.selector);
    }
    return max;
   }
   return max;
  }
  ...
}
```

Support Libraries

CML Domain Model Transformation

State Variables

Constraint Check

Derived Clause Constraints

Derived Clause Constraints

Listing 2. Generated Solidity contract from CML defintion in Listing 1.

TABLE II
MAPPING OF CML CONSTRUCTS TO SOLIDITY CONSTRUCTS

CML Construct	Solidity Construct
Party	Struct
Asset	Struct
Transaction	Struct
Enumeration	Enumeration
Event	Function
Function	Function
Top level function	Function with pure/view declaration
Clause constraints	Function modifier with conditional checks

B. Code Generation Idioms

1) Design Patterns: Design patterns [16], [17] are a commonly used technique to encode design guidelines or best practices. In previous work [18], [19] we have gathered design patterns for smart contracts in the Ethereum ecosystem along with corresponding code building blocks for Solidity, which can be directly integrated in the automatic code generation process. This procedure is exemplified with the “Ownership” and “PullPayment” pattern. The “Ownership” pattern satisfies a contract has an owner (by default the creator of a contract) and is used to limit access to sensitive functions to only that owner. The “PullPayment” pattern is used to mitigate security risks when sending funds by switching from a push to a pull payment, meaning that funds must be proactively withdrawn by the recipient. Listing 3 illustrates the application of patterns in CML and Listing 4 shows the generated code output.

```
namespace cml.examples

import cml.generator.annotation.solidity.*

@Ownership @PullPayment
contract BecomeRichest
   Party richest
   Integer mostSent

   clause BecomeRichest
      party anyone
      may becomeRichest

   action Boolean becomeRichest(TokenTransaction t)
      caller.deposit(t.amount)
      if(t.amount > mostSent)
         transfer(richest, token.quantity)
         richest = caller
         mostSent = t.amount
         return true
      return false
```

Listing 3. CML contract with design pattern annotation for the “Ownership” and “Pullpayment” pattern.

```
pragma solidity >=0.4.22 <0.7.0;
  ...
import "./lib/openzeppelin/Ownable.sol";
import "./lib/openzeppelin/PullPayment.sol";
  ...

contract BecomeRichest is ConditionalContract, Ownable, PullPayment {
  ...
 function becomeRichest() public payable
   checkAllowed("BecomeRichest")
   returns (bool)
  {
   if (msg.value > mostSent)
   {
    _asyncTransfer(richest.id , address(this).balance);
    richest = Party(msg.sender);
    mostSent = msg.value;
    return (true);
   }
   return (false);
  ...
}
```

Pattern Integration
Asynchronous Payment

Listing 4. An excerpt of the generated Solidity contract from Listing 3 utilizing design patterns.

2) Avoiding Overflows/Underflows: Signed and unsigned integers in Solidity are restricted in size to a range of values. For example, an unsigned 8-bit integer (uint8) may incarnate values between 0 and $255-$ $(2^8 - 1)$. If the result of an operation is outside of this supportd range an overflow or underflow occurs and the result is truncated. To illustrate this behavior, when using 8-bit unsigned integers, $255 + 1 = 0$. This result is more apparent in binary representation, where $1111\ 1111_2 + 0000\ 0001_2$ should result in $1\ 0000\ 0000_2$. However, since only 8 bits are available, the leftmost bit is lost, resulting in a value of $0000\ 0000_2$. These overflows can have serious consequences that one should mitigate against. One approach is to use `require` to limit the size of inputs to a reasonable range, or use a library for secure smart contract development like OpenZeppelin’s [20] “SafeMath”, to cause a revert for all overflows. The annotation `@SafeMath` on top of a CML contract adheres to the latter approach and automatically replaces all occurrences of arithmetic operations with equivalent “SafeMath” library calls, as shown in Listings 5 and 6.

```
namespace cml.examples

import cml.generator.annotation.solidity.*

@SafeMath
contract Counter
   Integer counter = 0

   clause ChangeCounter
      party anyone
      may increaseCounter or decreaseCounter

   action increaseCounter()
      counter = counter + 1

   action decreaseCounter()
      counter = counter - 1
```

Listing 5. CML contract with “SafeMath” annotation to indicate that arithmetic operations should be checked for overflows and underflows.

```
pragma solidity >=0.4.22 <0.7.0;
  ...
import "./lib/openzeppelin/SafeMath.sol";
  ...
contract Counter is ConditionalContract {
  ...
 uint counter = 0;
  ...
 function decreaseCounter() public
  checkAllowed("ChangeCounter")
 {
  counter = SafeMath.sub(counter, 1);
 }
  ...
}
```

Safe Arithmetics

Listing 6. An excerpt of the generated Solidity contract from Listing 5, containing wrapper calls for safe arithmetic operations.

3) Fixed Point Arithmetic: Solidity supports integer numbers, but decimal numbers are not yet supported. Although it is possible to declare fixed point number types, they cannot be assigned to or from. When dealing with decimals on systems that support only integers, fixed point arithmetic can be used. This is a technique for performing operations on numbers with fractional parts using integers. The approach builds on scaling an integer so that a certain (fixed) number of decimals are included, e.g. the value 1.23 can be represented as 123 with a scaling factor of $1/100$. In other words, the decimal values are “normalized” to integer values. Arithmetic operations are then executed on the underlying integers with the overhead of taking the scaling factors into account. The approach is

demonstrated in Listings 7 and 8. It should be noted that during the interaction with the Solidity contract, the input and output number values are of fixed-point type and require conversion in respect to the choosen scaling value.

```
namespace cml.examples

import cml.generator.annotation.solidity.*

def Integer equation()
   return 8 / 2 * (2 + 2)

@FixedPointArithmetic(decimals=2)
contract FixedPointArithmetic

   clause Clause
      party anyone
      may calc1 or calc2

   action Integer calc1()
      return equation() / 2

   action Real calc2()
      return equation().toReal() * 2.5
```

Listing 7. CML contract containing arithmetic operations and “FixedPointArithmetic” annotation.

```
pragma solidity >=0.4.22 <0.7.0;
  ...
import "./lib/cml/FPMath.sol";
  ...
contract FixedPointArithmetic is ConditionalContract {
  ...
  function calc2() public
    checkAllowed("Clause")
    returns (uint)
  {
    return (FPMath.fpmul(IntLib.toReal(equation()), 2.5E2, 2));
  }
  ...
  function equation() public pure
    returns (uint)
  {
    return (FPMath.fpmul(FPMath.fpdiv(8E2, 2E2, 2), (FPMath.add(2E2, 2E2)), 2));
  }
  ...
}
```

Fixed Point Conversion

Fixed Point Arithmetic

Listing 8. An excerpt of the generated Solidity contract from Listing 7, with applied fixed point conversion and fixed point arithmetic wrapper calls.

4) Type Collections: Solidity supports the concept of arrays and mappings (dictionaries). Mappings can be seen as hash tables which are virtually initialized such that every possible key exists and is mapped to a value whose byte-representation is all zeros: a type’s default value [21]. This has the drawback that mappings cannot be directly iterated over since there is no way to know how many keys exist, because they all exist. A common pattern is therefore to use an auxiliary array in combination with mappings to hold the keys that exist.

Collections of values in CML are denoted with [] after a type declaration and are transformed to a Solidity mapping, where the type identifier is used as key. Library code is generated for each collection containing a mapping and keystore to provide basic editing and iteration functionality for the collection. Due to missing generics in Solidity, this code must be dynamically created for each mapping to accommodate for different types. In order to minimize the operational complexity of key lookups, a circular linked list is used instead of an array to track mappings that exist. Key existence is checked by verifying that a key node has a valid pointer to the previous and next node, thus iterating all key entries can be avoided. The usage of a circular linked list has also the advantage to support collection implementation variations, e.g. key ordering, a FILO stack, or a FIFO ring buffer. Listings 9 to 11 illustrate the described approach.

```
namespace cml.examples

asset Asset identified by inventoryNumber
   Integer inventoryNumber
   Integer aquisitionCost

contract Mapping
   Asset[] assets

   clause AssetInteraction
      party anyone
      may addAsset or removeAsset or countAssets or countValuableAssets

   action addAsset(Asset a)
      if (!assets.contains(a.inventoryNumber))
         assets.add(a)

   action removeAsset(Integer inventoryNumber)
         assets.rmv(inventoryNumber)

   action Integer countAssets()
      return assets.size()

   action Integer countValuableAssets()
      var Integer count = 0
      for (a in assets)
         if (a.aquisitionCost > 100)
            count++
      return count
```

Listing 9. CML contract using a collection.

```
pragma solidity >=0.4.22 <0.7.0;
  ...
import "./lib/cml/Model.sol";
import "./lib/cml/MapUintAsset.sol";
  ...
contract Mapping is ConditionalContract {
  ...
  using MapUintAsset for MapUintAsset.Data;
  MapUintAsset.Data internal assets;
  ...
  function addAsset(Model.Asset memory a) public
    checkAllowed("AssetInteraction")
  {
    if ( !assets.contains(a.inventoryNumber))
    {
      assets.add(a.inventoryNumber, a);
    }
  }
  ...
  function countValuableAssets() public
    checkAllowed("AssetInteraction")
    returns (uint)
  {
    uint count = 0;
    for (uint i = 0; i < assets.size(); i++)
    {
      Model.Asset storage a = assets.getEntry(i);
      if (a.aquisitionCost > 100)
      {
        count++;
      }
    }
    return (count);
  }
```

Library Usage

Streamlined Type & Operations

Operation Usage

Listing 10. An excerpt of the generated Solidity contract from Listing 9.

```
pragma solidity >=0.4.22 <0.7.0;
  ...
import "./CLLUint.sol";
import "./Model.sol";
  ...
library MapUintAsset {
  ...
  struct Data {
    mapping(uint => Model.Asset) map;
    CLLUint.CLL mapIdList;
  }
  ...
  using CLLUint for CLLUint.CLL;
  ...
  function size(Data storage self) public view returns (uint) {
    return self.mapIdList.sizeOf();
  }
  ...
}
```

Mapping

Circular Linked List Key Storage

CLL Library Usage

Provided Operation

Listing 11. An excerpt of the generated Solidity collection library code.

VI. EVALUATION

In this section, we provide arguments to support the claim that our proposed language concepts lead, through a reduction of complexity, to an increased comprehensibility, and reduced error susceptibility. For this purpose we compare several contract use case scenarios (partly taken from the Solidity documentation) specified in CML with their respective Solidity implementations generated by our framework.

Solidity is a Turing complete language, which gives expressiveness and power, but can lead to less comprehensible code and a more difficult assessment of correctness. As a general rule, the more expressive a language is, the higher its complexity, and the more it is prone to include bugs and errors. In comparison with Solidity, our DSL contains abstraction much closer to the target domain, with the aim to promote clarity and comprehensibility.

In our context, to measure the complexity, we follow the characterization of complexity as detail complexity, defined by Senge [22] as "the sort of complexity in which there are many variables." We relate this definition to measuring the logical lines of code and syntactic elements (AST nodes) that are contained in respective representations. The number of AST nodes is determined for Solidity with the help of an ANTLR parser [23] (taking into account all "ASTNode" expressions), whilst for CML it is regarded as the elements contained in the generated AST (parse tree). Please note that this is not intended as a precise and generalizable measurement, but rather to give a rough comparison of our approach compared to contracts encoded directly in Solidity. Looking at the results, which are summarized in Table III, we can see that both metrics for the CML representation are always lower. On average CML performs 840% better in terms of LLOC and 610% better in terms of syntactic elements. Our results indicate that, in our examples, evidence for the higher abstraction level of CML can be found, which leads to lower complexity and in turn should lead to less susceptibility to errors.

TABLE III
COMPLEXITY COMPARISON BETWEEN CML REPRESENTATION AND GENERATED SOLIDITY IMPLEMENTATION

Use Case	CML LLOC[1]	CML SE[2]	Solidity LLOC	Solidity SE	Δ% LLOC	Δ% SE
Become Richest	18	43	208	502	1056	1067
Purchase	39	116	216	584	454	403
Simple Auction	28	75	402	558	1336	644
Time Lock	27	70	344	475	1174	579
Voting	64	227	485	1655	658	629
AVG	35,2	106,2	331,0	754,8	840,3	610,7

[1]Logical Lines of Code [2]Syntactic Element: AST Node

VII. DISCUSSION

There are several challenges regarding the formalization of contracts as discussed by Pace and Schneider [24]. A survey of formal languages for contracts performed by Hvitved [25] gives an overview of possible approaches. With regard to contract formalization, one of the main problems is the succinct, consistent, and sufficient representation of contractual statements. Finding the right abstractions for frequently recurring components in legal contracts that are relevant for the description of smart contracts is crucial. On this basis, a generic description covering a wide range of contract scenarios can be worked out. To our best knowledge no literature exists that deals with the conceptual analysis of relevant components. It rather seems that each work in the contract formalization field has its own assumptions about an optimal description.

Our approach of having clause statements, in addition to the imperative declaration of actions, is based on the idea of providing an abstract level of description that is closer to conventional contract clauses and provides a general overview of contract behavior. This has the advantage of better isolating execution context requirements for actions, which promotes comprehensibility and leaner actions, as only requirements depending on input parameters need to be checked within respective actions. To further extend the abstraction efforts, language constituents are used that are more closely related to application domain concepts, which helps to make their intended purpose more explicit. As an example, the *Party* and *Duration* type specifiers have a natural language meaning that is also accessible to non-programmers, as opposed to their corresponding representation as `address` and `uint` type in Solidity. CML builds a generic framework for contract formalization without an exuberant syntax leaking implementation details. The code generation process allows to alter abstracted contract specifics to an desired implementation form. Hence, a contract representation that is very compact can translate into a more verbose implementation language, in which the generator helps to construct the needed bulk.

Assuming that experts create the code generator, less experienced users can rely on a correct implementation. No manual coding effort is required, therefore accelerating the development time while decreasing the chance of errors when compared to manual coding from requirements. This is an important aspect, as there are many potential causes of programming errors in Solidity, like integer overflow and underflow, reentrancy, or timestamp dependence, to just name a few [26]. Further, in case of a required adaption to new best practices, the code generator routines must only be updated in one place and can be reused to generate adapted code.

On the negative side, since abstraction is always connected to information loss, a code generator can hardly cover all cases and it might be required to alter the generated output to inject some custom code. Another problem is that code is generated with the goal of being generally applicable, which entails increased complexity. This might result in generated code that is more elaborate and less comprehensible. In the context of Solidity this may mean that the code efficiency in terms of transaction costs is not on par with use case optimized code. As the financial aspect is an important concern in Solidity, this might hinder the adoption of this approach, but could be out

weight by the gained advantages concerning productivity and code quality.

Overall, if applied correctly, abstraction and code generation can increase the efficiency, clarity, and flexibility of code whilst reducing the susceptibility to errors.

VIII. Related Work

Several works pursue the approach to utilize a domain specific language and the concept of abstraction to facilitate the creation process of smart contracts.

Regnath and Steinhorst [27] have derived several language design concepts to approach a unified contract language demonstrated by a prototype implementation called SmaCoNat. The proposed concepts include the reliance on a small set of predefined operations and data types, an enforced sectioned code structure, limited aliasing, and building on natural language identifiers. Hence, approaching a unified contract language that enables a common understanding of code semantics on higher abstraction layers. In comparison to our work, the abstraction is put on a relatively high-level, which can pose limitations on the expressiveness of the language (as being too generic).

Another approach by Frantz and Nowostawski [28] proposes a semi-automated method for the translation of institutional constructs in a human readable behaviour specification to Solidity smart contracts. The applied conceptual approach is closely related to our work, in the sense that the institutional constructs describe the parties stipulations in a structured manner, similar to our clause formalization. Another commonality is the generation of Solidity code from an abstract representation, but unlike our work, the generated smart contracts contain only skeleton code and require considerable manual input to make them executable.

Yet another paper by He, Qin, Zhu, *et al.* [29] proposes a specification language for smart contracts called SPESC, which can define the specification of smart contracts for the purpose of collaborative design. The SPESC language contains term constructs that are akin to the clause constructs in our work, but the details of actions cannot be adequately specified and code generation is missing.

In general, the above mentioned publications contain interesting approaches and findings, but the proposed domain specific languages lack a model transformation to an executable smart contract implementation, which is demonstrated in this work. To our best knowledge, apart from our and the already mentioned work by Frantz and Nowostawski [28], there is only one further work by [30] that deals with the generation of Solidity code from an abstract representation. Regarding the clause formalization itself, our approach can be compared with works by Prisacariu and Schneider [31] and Martínez, Díaz, Cambronero, *et al.* [32] in which contract specification is based on the deontic notions of obligation, permission, and prohibition applied to actions. This approach is often used in the formalization of contracts.

In order to also point out efforts from the legal tech industry in regards to contract specific DSLs, the Accord Project [33] is to be mentioned. It is an open source, non-profit initiative developing specifications and open-source software tools for future smart legal contracting. The project aims to provide an open, standardized format for smart legal contracts that binds legally enforceable text in natural language to executable business logic. The proposed toolchain contains Ergo [34], a DSL with which the execution logic of legal contracts can be specified. The language features programming constructs specifically designed for legal contracts, thus it is also comparable to CML.

IX. Conclusion

In this paper we have analysed important contract building blocks and proposed a high level smart contract language called Contract Modeling Language (CML). CML incorporates a fluently readable, clause like formalization concept to describe the individual operational intents (commitments) of contract participants. This approach enables a representation that is conceptually and syntactically easier to grasp and thus also improves reasoning about a contract. Another key point is that CML describes contract semantics on an higher level and transfers the specifics of an implementation to lower levels. Consequently, the specification of a contract with its underlying model and defined behavior can be decoupled from the actual implementation. This aspect is demonstrated by transforming contracts from CML to Solidity code. It is possible to automate platform specific implementation steps, for example the inclusion of design patterns or coding abstractions. Thus, contract creators can be shielded from low level implementation specific tasks.

For future work, we plan to evaluate the efficiency of our approach in an experiment. Further, we plan to enhance the Solidity code generation process, with other commonly occurring design patterns, coding abstractions, and more powerful code inference mechanisms. Beyond that, code generation support for another smart contract platform can be incorporated. This could give more insights about the general applicability of the proposed smart contract abstraction mechanisms, leading to future improvements and extensions.

References

[1] H. E. Willis, “Restatement of the Law of Contracts of the American Law Institute,” *Ind. LJ*, vol. 7, p. 429, 1931.

[2] N. Szabo. (1994). Smart Contracts, [Online]. Available: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html (visited on 09/10/2019).

[3] ——, (1997). The idea of smart contracts, [Online]. Available: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_idea.html (visited on 09/10/2019).

[4] M. K. Woebbeking, “The Impact of Smart Contracts on Traditional Concepts of Contract Law,” no. 1988, pp. 106–113, 2019.

[5] G. Wood, “Ethereum: a secure decentralised generalised transaction ledger,” *Ethereum Proj. Yellow Pap.*, vol. 151, pp. 1–32, 2014.

[6] M. Fowler and R. Parsons, *Domain-Specific Languages*. 2010.

[7] L. Johnson, “Effective Contract Drafting: Indentifying the Building Blocks of Contracts,” *Sch. Work.*, Jan. 2013.

[8] A. R. Hevner, S. T. March, J. Park, and S. Ram, “Design science in information systems research,” *MIS Q. Manag. Inf. Syst.*, 2004.

[9] R. J. Wieringa, *Design science methodology: For information systems and software engineering*. 2014.
[10] Xtext framework, [Online]. Available: https://www.eclipse.org/Xtext/ (visited on 07/10/2019).
[11] Contract Modeling Language, [Online]. Available: https://github.com/maxwoe/cml.
[12] CML Web Editor, [Online]. Available: http://cml.swa.univie.ac.at/.
[13] O. Marjanovic and Z. Milosevic, "Towards Formal Modeling of e-Contracts," *Proc. Fifth IEEE Int. Enterp. Distrib. Object Comput. Conf.*, pp. 59–68,
[14] J. D. Kruijff and H. Weigand, "On the Move to Meaningful Internet Systems. OTM 2017 Conferences," vol. 10573, pp. 383–398, 2017.
[15] D. McAdams, "An Ontology for Smart Contracts," *IOHK Pap.*, p. 3, 2017.
[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996, p. 395.
[17] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Chichester, UK: Wiley, 2000.
[18] M. Wohrer and U. Zdun, "Smart contracts: Security patterns in the ethereum ecosystem and solidity," *2018 IEEE 1st Int. Work. Blockchain Oriented Softw. Eng. IWBOSE 2018 - Proc.*, vol. 2018-Janua, pp. 2–8, 2018.
[19] ——, "Design Patterns for Smart Contracts in the Ethereum Ecosystem," in *2018 IEEE Int. Conf. Internet Things*, 2018, pp. 1513–1520.
[20] OpenZeppelin. OpenZeppelin/zeppelin-solidity: OpenZeppelin, a framework to build secure smart contracts on Ethereum, [Online]. Available: https://github.com/OpenZeppelin/zeppelin-solidity (visited on 12/05/2017).
[21] Solidity 0.5.13 documentation, [Online]. Available: https://solidity.readthedocs.io/en/v0.5.13/ (visited on 11/25/2019).
[22] P. M. Senge, *The fifth discipline: The art and practice of the learning organization*. Broadway Business, 2006.
[23] GitHub - federicobond/solidity-parser-antlr: A Solidity parser for JS built on top of a robust ANTLR4 grammar, [Online]. Available: https://github.com/federicobond/solidity-parser-antlr (visited on 02/20/2020).
[24] G. J. Pace and G. Schneider, "Challenges in the Specification of Full Contracts," in, Springer, Berlin, Heidelberg, 2009, pp. 292–306.
[25] T. Hvitved, "Contract Formalisation and Modular Implementation of Domain-Specific Languages,"
[26] Known Attacks - Ethereum Smart Contract Best Practices, [Online]. Available: https://consensys.github.io/smart-contract-best-practices/known_attacks/ (visited on 12/03/2019).
[27] E. Regnath and S. Steinhorst, "SmaCoNat: Smart Contracts in Natural Language," *Forum Specif. Des. Lang.*, vol. 2018-Septe, 2018.
[28] C. K. Frantz and M. Nowostawski, "From institutions to code: Towards automated generation of smart contracts," *Proc. - IEEE 1st Int. Work. Found. Appl. Self-Systems, FAS-W 2016*, pp. 210–215, 2016.
[29] X. He, B. Qin, Y. Zhu, X. Chen, and Y. Liu, "SPESC: A Specification Language for Smart Contracts," *Proc. - Int. Comput. Softw. Appl. Conf.*, vol. 1, pp. 132–137, Jul. 2018.
[30] A. Mavridou and A. Laszka, "Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach," *arXiv Prepr. arXiv1711.09327*, 2017.
[31] C. Prisacariu and G. Schneider, "A Formal Language for Electronic Contracts," *Lect. Notes Comput. Sci.*, vol. 4468, pp. 174–189, 2007.
[32] E. Martínez, G. Díaz, E. Cambronero, and G. Schneider, "A model for visual specification of e-contracts," *Proc. - 2010 IEEE 7th Int. Conf. Serv. Comput. SCC 2010*, vol. 8625 LNAI, no. section 3, pp. 1–8, Jul. 2010.
[33] Accord Project, [Online]. Available: https://www.accordproject.org/ (visited on 02/20/2020).
[34] Ergo - Accord Project, [Online]. Available: https://www.accordproject.org/projects/ergo/ (visited on 02/20/2020).