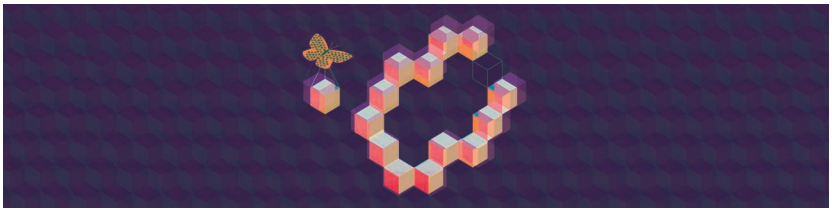
From Domain-Specific Language to Code: Smart Contracts and the Application of Design Patterns

Maximilian Wöhrer and Uwe Zdun, University of Vienna

// *Due to the conceptual discrepancy between legal contract terms and code, understanding and creating smart contracts without errors can be difficult. This article proposes smart contract design patterns and their automated application through (the use of) a domain-specific language and code generation.*//



SMART CONTRACTS BASED on blockchain technologies have received a lot of attention in the recent past. Their potential to open up new business applications by replacing the established trust concept of intermediaries sparked a hype around them. However, the initial euphoria was dampened by serious security incidents[1] and the insight that failing contracts can entail huge financial losses. This applies in particular to one of the most prominent implementation platforms for smart contracts, Ethereum. Ethereum is an open source, distributed computing platform allowing the creation and execution of decentralized programs (smart contracts) in its own blockchain.[2] Nowadays, creating such smart contracts that convert legal contracts into a machine-readable and executable form is a challenging task. Beyond the conceptual gap between natural (legalese) language and equivalent code, various peculiarities of the underlying blockchain environment complicate the correct and secure creation of smart contracts even further.[3] These include the fixed and autonomous nature of code execution, the lack of high-level coding abstractions, and the rapid progress of the development framework. In view of these problems, it is beneficial to have a foundation of established design and coding guidelines along with a framework for their application. To pursue this goal, it is advantageous to base smart contracts on higher-level, well-established designs that have emerged as best practices.[4] To this end, we propose design patterns[5] for creating smart contracts in the context of Ethereum and similar platforms. To automate the application of these design patterns and to avoid errors due to their manual coding, we also propose a DSL for smart contracts and a code generator to generate Solidity code from the DSL. A DSL is a computer programming language of limited expressiveness focused on a particular application domain.[6]

Digital Object Identifier 10.1109/MS.2020.2993470
Date of current version: 20 August 2020

Contract Stages

Contracts are usually preceded by an abstract agreement that specifies elementary modalities and actions between parties. To avoid the ambiguity of natural language and to explicate terms and conditions as clearly and completely as possible, contracts are written in legalese. This can be understood as a first formalization method. A machine-readable representation, usually in a formal language, is retrieved from a conversion of the traditionally written contract, although a contract

could be immediately specified in a formal language. A written contract is grounded on law, where enforceability is considered to be *ex post*, i.e., a party can enforce a settlement at court only after a contractual breach. This stands in contrast to the formal machine representation and its realization as a smart contract, where the execution is based on an architecture that by design does not allow nonconformism; hence enforceability is considered to be *ex ante*. The above-described principles are depicted in Figure 1, which gives an overview of the different contract stages, namely negotiation, formation, and performance.

> It is advantageous to base smart contracts on higher-level, well-established designs that have emerged as best practices.

Smart Contract Design Patterns

Design patterns express an abstract or conceptual solution to a concrete, complex, and reoccurring problem. In the context of Ethereum-based smart contract development we have elaborated several design patterns (see Wöhrer and Zdun[7,8] for details). The patterns help to solve commonly recurring application requirements and address common problems and vulnerabilities connected to smart contract design. According to their operational scope, they are divided into five categories: 1) action and control, 2) authorization, 3) lifecycle, 4) maintenance, and 5) security. Although some patterns are very basic, their real practical value unfolds when patterns are used as a prerequisite or in combination with other patterns. Table 1 provides an overview of the pattern categories and associated patterns, including a brief description of the underlying problem and its solution. More details about the patterns along with Solidity coding practices can be studied in Wöhrer and Zdun.[7,8]

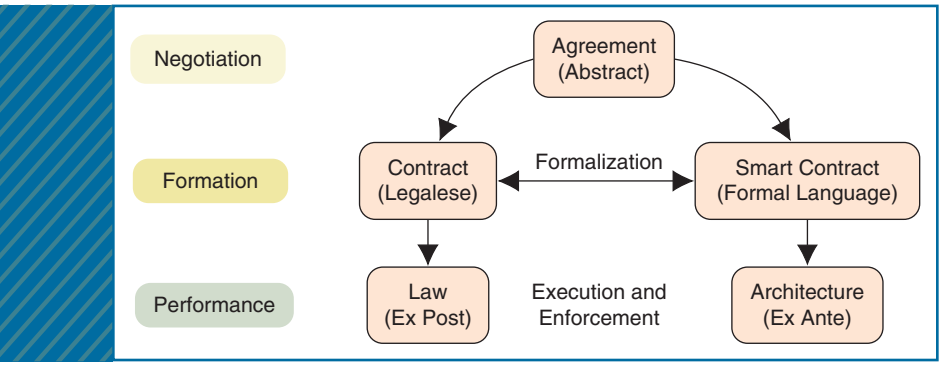


FIGURE 1. The contract stages and conceptual relation between traditional and smart contracts.

DSLs for Smart Contract Development

DSLs can express a problem discipline in a more natural way for domain experts, making the entire development process more efficient and less error prone. Several smart contract DSLs exist with various language concepts and programming paradigms, like DAML,[9] a functional language influenced by Haskell; Ergo,[10] an imperative language; and Archetype,[11] a declarative and imperative language focusing on formal verification, to just name a few. In contrast to these languages, we propose a DSL called *Contract Modeling Language* (*CML*)[12] that builds on a clause grammar close to natural language to mimic the obligations expressed in contracts. The language is implemented in Xtext[13] and available on GitHub.[14]

CML

CML is a high-level DSL using a declarative and imperative formalization as well as object-oriented abstractions to specify smart contracts. As seen in the exemplary CML contract in Figure 2, contracts in CML consist of state variables (lines 7–10) and actions (lines 22–35), which operate on these. In addition, contracts contain clause statements (lines 12–20) that resemble natural language to mimic and capture contractual obligations of involved parties. More precisely, these statements represent so-called covenants, which are specific contractual clauses that belong to elementary building blocks of contracts and enclose promises to engage in or refrain from certain

actions. The conceptual structure of clause statements (Figure 1, lower left) is derived from an analysis of typical covenant components and looks as follows. Each clause statement must specify at least an actor, an action, and the modality of that action ("may" or "must") and has a unique identifier for referencing. Optional elements include temporal or state constraints. Temporal constrains are indicated by the keyword "due" followed by a temporal precedence (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 derived. This includes the performance of a clause, the execution of an action, or the occurrence of an external event. General constraints can be defined after the keyword "given" by multiple linked conditions, which usually refer to the contract state. In general, the described clause statements support a more natural and practical contract representation compared to an unstructured code implementation and offer a better overview of contract behavior at a glance.

To further expand efforts toward abstraction, CML uses a type system

Table 1. An overview of smart contract design patterns.

Action and control	
Pull payment	As a send operation can fail, let the receiver withdraw the payment.
State machine	When different contract stages are needed, these are modeled and represented by a state machine.
Commit and reveal	As blockchain data are public, a commitment scheme ensures confidentiality of contract interactions.
Oracle (data provider)	When knowledge outside the blockchain is required, an oracle pushes information into the network.
Authorization	
Ownership	As anyone can call a contract method, restrict the execution to the contract owner's address.
Access restriction	When function execution checkups are needed, these are handled by generally applicable modifiers.
Lifecycle	
Mortal	Since deployed contracts do not expire, self-destruction with a preliminary authorization check is used.
Automatic deprecation	When functions shall become deprecated, apply function modifiers to disable their future execution.
Maintenance	
Data segregation	As data and logic usually reside in the same contract, avoid data migration on updates by decoupling.
Satellite	As contracts are immutable, functions that are likely to change are outsourced into separate contracts.
Contract register	When the latest contract version is unknown, participants proactively query a register.
Contract relay	When the latest contract version is unknown, participants interact with a proxy contract.
Security	
Checks-effects-interaction	As calls to other contracts hand over control, avoid security issues by a functional code order.
Emergency stop	Since contracts are executed autonomously, sensitive functions include a halt in the case of bugs.
Speed bump	When task execution by a huge number of users is unwanted, prolong completion for counter measures.
Rate limit	When a request rush on a task is not desired, regulate how often a task can be executed within a period.
Mutex	As reentrancy attacks can manipulate contract state, a mutex hinders external calls from reentering.
Balance limit	There is always a risk that a contract gets compromised; limit the maximum amount of funds held.

that is more closely related to application domain concepts. CML offers not only typical primitive types (Boolean, String, and so forth), but also basic temporal types (DateTime, Duration), and easily extensible structural composite types (Party, Asset, Transaction, Event) to embody common contract-specific concepts and operations. Through these measures a generic framework for contract formalization is provided, that avoids an excessive syntax containing implementation specifics. As a result, higher-level contract specifications can be translated into a desired form of implementation, leading to a decoupling of contract specification and implementation.

CML Code Generation to Solidity

As proof of concept, we have implemented a code generator that assigns CML abstractions to appropriate Solidity equivalents. During this procedure, design patterns and other practical coding idioms can be applied. To illustrate this process, Figure 2 shows a simple auction contract in CML and the correspondingly generated output in Solidity. The generator traverses the CML representation to produce Solidity code that relies on static and dynamically created support libraries. These libraries contain the implementation of CML abstractions. As such, they embody CML-type operations (line 4) or refer to established libraries for smart contract development (line 5). In light of this, type

CML

```
1  namespace cml.examples
2
3  import cml.generator.annotation.solidity.*
4
5  @PullPayment
6  contract SimpleAuction
7     Integer highestBid
8     Party highestBidder
9     Party beneficiary
10    Duration biddingTime
11
12    clause Bid
13       due within biddingTime after contractStart
14       party anyone
15       may bid
16
17    clause AuctionEnd
18       due after contractStart.addDuration(
               biddingTime)
19       party beneficiary
20       may endAuction
21
22    action init(Duration _biddingTime, Party
           _beneficiary)
23       biddingTime = _biddingTime
24       beneficiary = _beneficiary
25
26    action bid(TokenTransaction t)
27       ensure(t.amount > highestBid, "There
              already is a higher bid.")
28       caller.deposit(t.amount)
29       if (highestBid != 0)
30          transfer(highestBidder, highestBid)
31       highestBidder = caller
32       highestBid = t.amount
33
34    action endAuction()
35       transfer(beneficiary, highestBid)


Conceptual structure of a clause statement:

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

Trigger:      AT | ClauseTrig | EventTrig | ActionTrig
ClauseTrig:   clause ID (fulfilled|failed)
ActionTrig:   ACTOR did ACTION
EventTrig:    event ID

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

(a)

Solidity

```
1  pragma solidity >=0.4.22 <0.7.0;
2  pragma experimental ABIEncoderV2;
3  import "./lib/cml/ConditionalContract.sol";
4  import "./lib/cml/DateTime.sol";
5  import "./lib/openzeppelin/PullPayment.sol";
6    ...
9  contract SimpleAuction is ConditionalContract, PullPayment {
10
11   struct Party {
12     address payable id;
13   }
14
15   uint highestBid;
16   Party highestBidder;
17   Party beneficiary;
18   uint biddingTime;
19   uint _contractStart;
20
21   constructor(uint _biddingTime, Party memory _beneficiary) public {
22     biddingTime = _biddingTime;
23     beneficiary = _beneficiary;
24     _contractStart = now;
25   }
26
27   function bid() public payable checkAllowed("Bid") {
28     require(msg.value > highestBid, "There already is a higher bid.");
29     if (highestBid != 0) {
30       _asyncTransfer(highestBidder.id , highestBid);
31     }
32     highestBidder = Party(msg.sender);
33     highestBid = msg.value;
34   }
35
36   function endAuction() public checkAllowed("AuctionEnd") {
37     _asyncTransfer(beneficiary.id , highestBid);
38   }
39    ...
42   function clauseAllowed(bytes32 _clauseId) internal returns (bool) {
43     if (_clauseId == "Bid") {
44       require(onlyAfter(_contractStart, biddingTime, true), "Function not called
               within expected timeframe");
45       return true;
46     }
47     if (_clauseId == "AuctionEnd") {
48       require(onlyBy(beneficiary.id), "Caller not authorized");
49       require(onlyAfter(DateTime.addDuration(_contractStart, biddingTime), 0,
               false), "Function called too early");
50       return true;
51     }
52     return false;
53   }
54    ...
76 }
```

Support Libraries

CML Domain Model Transformation

State Variables

Clause Constraint Check

Pull Payment Pattern

Derived Clause Constraints

Derived Clause Constraints

(b)

FIGURE 2. An auction contract (a) specified in CML and (b) showing an excerpt of the generated Solidity code.

operations in CML are merely declared as method signatures and are implemented in Solidity through library calls. Regarding the structural transformation, CML types are mapped to conceptual Solidity type equivalents (lines 11–19), actions are converted to Solidity functions (lines 21–38), and clause statements are transformed into declarative checks (lines 43–52) that are applied by the `checkAllowed` modifier (lines 27 and 36) added to those functions. The modifier itself is inherited from `ConditionalContract` and contains a call to `clauseAllowed` (overridden in line 42) as well as code to track the context (e.g., time, caller) of successfully executed functions. During code generation it is also possible to apply design patterns. In the depicted example the Pull Payment pattern is applied to mitigate security risks when sending funds by switching from a push to a pull payment. An appropriate annotation instructs the generator to incorporate the pattern implementation (line 9) and engage asynchronous payments for all outgoing token transfers (line 30). Various other practical transformation schemes are possible. For example, an automatic application of wrapper calls for arithmetic operations to avoid type overflows and underflows. In addition, decimal numbers (although not currently supported by Solidity) can be used seamlessly through fixed point arithmetic, with the generator automatically inducing the appropriate value assignments and arithmetic calculations.

In this article, we have presented smart contract design patterns and proposed a high-level smart contract language called *CML*. The patterns provide guidance for addressing common smart contract design challenges in Ethereum, and the DSL provides useful abstractions for the specification of smart contracts. The combined application of both is shown by transferring a CML specification into a Solidity implementation that follows established design recommendations. The associated code generation serves to automate platform-specific implementation steps by mapping abstractions to suitable target equivalents and integrating useful design patterns and coding practices. The proposed approach can reduce the design complexity, since an abstract representation that is very compact and close to the target domain is translated into a more verbose and low-level implementation. In addition, the approach can reduce the susceptibility to errors, assuming the code generator generates correct code. Overall, the use of a DSL including code generation based on design patterns can increase the efficiency, clarity, and flexibility of smart contract development while reducing the susceptibility to errors.

ABOUT THE AUTHORS

MAXIMILIAN WÖHRER is a researcher at the Faculty of Computer Science, University of Vienna, Austria. His research interests include blockchain technology, smart contracts, software architecture, software engineering, and the application of design patterns in the aforementioned domains. Wöhrer received a master's degree in computer science in 2009 (with distinction) from the University of Vienna and is currently pursuing a Ph.D. in computer science from that same institution. Contact him at maximilian.woehrer@univie.ac.at.

UWE ZDUN is a full professor of software architecture at the Faculty of Computer Science, University of Vienna, Austria. His research interests include software design and architecture, empirical software engineering, distributed systems engineering, software patterns, domain-specific languages, and model-driven development. Zdun received a doctoral degree from the University of Essen in 2002. He has published more than 210 articles in peer-reviewed journals, conferences, book chapters, and workshops, and is the coauthor of several books. Contact him at uwe.zdun@univie.ac.at.

References

1. GitHub, "Major issues resulting in lost or stuck funds." Accessed on: Jan. 2, 2020. [Online]. Available: https://github.com/ethereum/wiki/wiki/Major-issues-resulting-in-lost-or-stuck-funds
2. Ethereum. Accessed on: Jan. 7, 2020. [Online]. Available: https://www.ethereum.org/

3. W. Dingman et al., "Defects and vulnerabilities in smart contracts, a classification using the NIST bugs framework," *Int. J. Networked Distributed Computing*, vol. 7, no. 3, pp. 121–132, 2019.
4. GitHub, "Ethereum smart contract best practices." Accessed on: Apr. 16, 2020. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/recommendations/
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Software*. Boston: Addison-Wesley Longman Publishing Co., Inc., 1996.
6. M. Fowler and R. Parsons, *Domain-Specific Languages*, 1st ed. Boston: Addison-Wesley Professional, 2010.
7. M. Wöhrer and U. Zdun, "Smart contracts: Security patterns in the ethereum ecosystem and solidity," in *Proc. IEEE 1st Int. Workshop Blockchain Oriented Software Eng. (IWBOSE)*, Campobasso, Italy, 2018, pp. 2–8. [Online]. Available: http://ieeexplore.ieee.org/document/8327565/
8. M. Wöhrer and U. Zdun, "Design patterns for smart contracts in the ethereum ecosystem," in *Proc. 2018 IEEE Int. Conf. Internet of Things (iThings)*, pp. 1513–1520. [Online]. Available: https://ieeexplore.ieee.org/document/8726782
9. DAML. Accessed on: Apr. 17, 2020. [Online]. Available: https://daml.com/
10. Accord Project, "Ergo." Accessed on: Feb. 20, 2020. [Online]. Available: https://www.accordproject.org/projects/ergo/
11. Archetype, "What is Archetype." Accessed on: Apr. 17, 2020. [Online]. Available: https://docs.archetype-lang.org/
12. M. Wöhrer and U. Zdun, "Domain specific language for smart contract development," in *Proc. 2020 IEEE Int. Conf. Blockchain and Cryptocurrency*. [Online]. Available: http://eprints.cs.univie.ac.at/6341/
13. Eclipse, "Xtext." Accessed on: July 10, 2019. [Online]. Available: https://www.eclipse.org/Xtext/
14. GitHub, "CML (contract modeling language)." Accessed on: Apr. 17, 2020. [Online]. Available: https://github.com/maxwoe/cml

Digital Object Identifier 10.1109/MS.2020.3012071