A Practical Notion of Liveness in Smart Contract Applications

Jonas Schiffl[1]
KASTEL - Institute of Information Security and Dependability, Karlsruhe Institute of Technology, Germany

Bernhard Beckert
KASTEL - Institute of Information Security and Dependability, Karlsruhe Institute of Technology, Germany

Abstract

Smart contracts are programs which manage resources in blockchain networks in an automated fashion. In this context, liveness properties are often essential: If I transfer money to a contract, will I eventually get it back?

This kind of property can be hard to specify and verify, in particular because application-specific fairness assumptions w.r.t. function invocation and the behavior of other parties are usually necessary for any liveness proof to succeed. In this work, we analyze smart contract liveness properties discussed in the literature. We find that the smart contract paradigm of decentralization and trustlessness implies that "real" liveness properties do not usually occur. The properties that have been classified as liveness can be more aptly described as *enabledness*, i.e., the ability of an agent to induce a state change, such as a transfer of resources.

Our contribution in this work is a specification language based on LTL to capture this kind of property. It features some special constructs to describe common properties in smart contracts, such as transfers or ownership of cryptocurrency. We show how often-used examples of liveness properties can be succinctly specified in our language. Moreover, we show how our notion of liveness can simplify formal verification compared to existing approaches.

2012 ACM Subject Classification Software and its engineering → Formal methods

Keywords and phrases Smart Contracts, Formal Verification, Security, Safety and Liveness

Digital Object Identifier 10.4230/OASIcs.FMBC.2024.8

Funding This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs.

1 Introduction

Smart contracts are programs which run in conjunction with blockchains. They typically manage resources, especially cryptocurrency. Abstractly, a smart contract application can be viewed as a set of functions and state variables. Actors in the network can call the functions and thereby change an application's state. Function calls are executed atomically in no pre-defined order.

Due to their unique characteristics, it is very important that smart contracts are correct upon deployment. In this work, we propose a novel perspective on an important and challenging class of correctness properties, namely *liveness*, in the context of smart contracts. In general, liveness properties can take many forms, depending on the application domain. One classic example is termination: Given a function, we may ask whether it always finishes

[1] Corresponding author

execution. In other domains, especially in distributed or parallel settings, deadlock freedom is essential: Is there always a way to continue execution, or is it possible to reach a situation where no progress can be made?

In this work, we argue that in the domain of smart contracts, liveness properties typically require that a certain functionality is (or becomes) accessible to an actor. In Section 2, we substantiate this intuition by analyzing the examples given in existing literature on smart contract liveness verification. In Section 3, we formalize our notion and develop a specification language for smart contract application liveness properties, based on a subset of LTL. We also sketch possibilities for verification. In Section 4, we demonstrate the use of the language on some examples from literature.

Related Work

Formal verification of smart contracts is a very active field of research. Most of the work targets smart contracts on the Ethereum platform. Many approaches focus on the detection of pre-defined vulnerabilities, which are detected by various kinds of static analysis. Recent overviews are given by He et al. [5] and Munir et al. [9].

Other tools also allow specification and verification of user-defined correctness properties. Recent versions of the Solidity compiler itself include an SMT-based tool that checks the validity of assertions [7], hinting at the high importance of formal verification for smart contract development. Other tools allow even more powerful properties to be specified and verified. In SOLC-VERIFY[4] and CELESTIAL [2], developers can specify invariants and function contracts consisting of first-order logic pre- and postconditions as well as frame conditions. VerX [12] introduces temporal operators and verification of safety properties.

Nam and Kil [10] present an approach for model-checking ATL properties of smart contracts by translating Solidity to the language of the MCMAS model checker. Their approach focuses on strategies and behaviors of actors, highlighting the need to consider different agents and their interests.

In the approach of Godoy et. al. [3], Solidity smart contracts are manually translated to the Alloy modelling language, providing an abstract, state transition based view of the smart contract application for visualization and auditing purposes. The approach works with the concept of enabledness of smart contract functions in the form of *enabledness preserving abstractions*. `requires` statements in the code determine whether a function is enabled, relating it to the notion of enabledness we develop in this work.

Furthermore, to our knowledge, there are two tools for specification and verification of liveness properties. Both are specific to the Solidity programming language. VeriSolid [8] is a tool for developing Solidity smart contracts through modeling them as state transition systems. The state of an application is modeled explicitly. Transitions are written directly in the supported Solidity subset. This model is translated into a BIP (Behavior, Interaction, Priority) model, which can be model-checked, e.g., for safety and liveness properties like deadlock freedom. VeriSolid allows specification of liveness properties in CTL. However, the properties that can be proven are concerned with successful termination after a function is called. There is no notion of fairness assumptions or the ability of an actor to effect a transfer.

SmartPulse [14] is a tool for checking safety and liveness properties of a given Solidity smart contract. Properties are specified in SmartLTL, which contains primitives for functions being called, functions finishing execution, reverting, and sending Ether (Ethereum's built-in cryptocurrency). Fairness assumptions can be specified if necessary to prove liveness properties. In addition to source code and property specifications, an environment is

specified, consisting of an attacker model (e.g., bounds on the number of re-entrant calls) and a blockchain model (e.g., gas costs of function calls). The SmartPulse paper provides a number of example liveness properties which we will discuss in the next section.

2 Liveness Properties in Smart Contracts

Smart contract platforms have several characteristics that influence what kind of liveness properties are important in a smart contract application. First, smart contracts exist in an open world: As a matter of principle, anyone can call any function and thereby trigger a transaction. Furthermore, smart contract platforms specifically exist for use cases where participants in the network do not necessarily trust each other. Therefore, participants generally cannot be assumed to behave in any particular way, at least in the absence of incentives.

A second defining characteristic of smart contracts is money: Most smart contract platforms have some form of cryptocurrency built in, and transferring currency or tokens is a part of almost all real-world smart contract applications. This means that there are usually financial incentives.

These characteristics lead to a special kind of liveness property which is highly common in smart contracts: "If I transfer money to a smart contract, will I get it back?" Or, more generally: Will some desired state change eventually happen? We elaborate on this kind of liveness property via a few simple examples, and demonstrate its pervasiveness by a brief review of example liveness properties in the literature.

2.1 Simple Bank Example

An example often used to showcase basic functionality is a simple smart contract version of a bank, which allows other accounts to deposit money, logs the balance of each account, and enables withdrawing funds according to the caller's balance (which is stored in a mapping `bals`). There are only two public functions, `deposit` and `withdraw`. They both have no precondition. The postcondition is that money is transferred from the caller to the bank contract (or vice versa for `withdraw`) and that the `bals` mapping is updated accordingly.

One major correctness property of this application can be viewed as a liveness property: If I deposit money in the bank, I will eventually get it back.

2.2 Escrow Example

Another common example is an escrow, where a smart contract application acts as an intermediary for a purchase[2]. For a successful purchase, the application proceeds through a succession of predefined states, according to the actions of buyer and seller. There are several liveness properties integral to the correctness of the application, depending on the exact implementation. For example, after the buyer confirms they received the purchased item, the seller should eventually be refunded their deposit.

[2] `https://docs.soliditylang.org/en/latest/solidity-by-example.html#safe-remote-purchase`

2.3 Auction Example

Another example[3] is a smart contract implementing an auction. Here, we consider an application consisting of a single contract which has three state variables: a boolean variable `state` which records the state of the auction (open, closed, or finalized), a mapping `bids` which records the bids made by each actor resp. account, and `highestBidder` which stores the current leader of the auction. Furthermore, the contract has four functions: `bid()` transfers some amount of currency (specified by the caller) from the caller to the auction contract. If the amount is higher than the current leading bid, the `highestBidder` variable is overwritten with the caller's address, and their bid is recorded in `bids`. The function `close()` sets the `state` variable to `closed` and then assigns ownership of the auction item to the current highest bidder. The function `claim` requires that the auction is closed, but not finalized. It transfers the amount of the winning bid to the auctioneer. Finally, `withdraw()` can be called by all losing bidders. It transfers the corresponding amount (as recorded in `bids`) to the caller.

Like in the bank example, a crucial correctness property of this application is a liveness property: If an actor makes a bid, that actor will eventually either win the auction and be assigned ownership of the desired item, or they will get their money back.

2.4 Examples from Literature

The SmartPulse paper [14] lists 23 safety and liveness properties of 10 applications that can be verified with their tool. Of these, 13 are liveness properties, signified by the *eventually* keyword. All of them fall into one of two categories: In the first category are properties that represent postconditions of a function, like the following: *"If a user withdraws funds after refunds are enabled, they will eventually be sent the sum of their deposits."* The paper on VeriSolid [11] only gives a single example of a liveness property, which also falls into this category.

The second category is of the type described in the beginning of this section, stating that some desired action will happen eventually. One of the examples in this category is an auction smart contract exactly like the one described above. Another example is the following statement about a crowdfunding application: *"If the campaign fails, the backers can eventually get their refund."*

In SmartPulse, liveness properties can be specified in a variant of LTL. Properties of the second category require a *fairness assumption* about the actors' behavior in order for verification to succeed: If a withdraw functionality is available, but never called, then losing bidders will not get their money back, although they could! The fairness assumption in this auction scenario is that any losing bidder will eventually call the `withdraw` function.

2.5 Observations

From the examples above, we note several important points. First, many liveness properties in smart contract applications can be reduced to postconditions and termination of a single function. Since there are several tools for specifying and verifying function postconditions, we focus on the other kind of liveness property, which states that a desired state change will happen eventually.

---

[3] used, e.g., in the documentation of the Solidity programming language: `https://docs.soliditylang.org/en/stable/solidity-by-example.html#simple-open-auction`

Concerning this kind of "real" liveness property, we observe that the crucial point about a desirable state change is whether an actor is able to effect it. There is a subtle difference: Liveness in smart contracts is not about whether something will definitely happen, but about whether someone can make it happen. In the auction example, what we want to prove is not whether every losing bidder actually gets their money back, it is that they *can* get it back (if they take the appropriate action).

This phrasing leads to the insight that in the context of smart contracts, it should be possible to specify liveness properties without having to specify assumptions regarding behavior, at all. What should be specified is *enabledness*, i.e., the ability to effect a desired result. This ability often pertains to a specific actor (or set of actors). Furthermore, on some examples, the desired change can only be effected after some fixed amount of time has elapsed, or if some other condition is met.

One last observation is that liveness is usually connected to resources, e.g., the built-in cryptocurrency of a blockchain network. Often, liveness corresponds to ownership: If an actor is able to effect a transfer of an amount of currency from a smart contract to themselves, then they own this amount, even though it is not stored in their own account.

3 Formalization of Smart Contract Liveness

In this section, we formalize the insights from the previous section and propose a practical way of specifying liveness properties for smart contract applications.

3.1 A Model of Smart Contract Applications

Our goal is a specification language that is independent of a concrete smart contract platform. Therefore, we assume a very generic model of smart contract applications (model elements in italics): First, we have a notion of *Accounts* identified by a name (e.g., an address). These can be either *External Accounts* (representing human actors) or *Contracts*. Each account has an integer-valued balance.

An application is a set of *Contracts*. Each *Contract* consists of a set of *Functions* and a set of *State Variables*. *State Variables* have a name and a type. The overall state of an application is determined by the values of all state variables (including account balances). The state only changes as a consequence of a function call, after a function has executed successfully.

Functions consist of a name, a set of parameters, and a set of return values. Each function call happens within a call context. For our purposes, this context consists of the account making the call, the amount of money transferred, and the list of parameter values at call time.

Crucially, each function also has a function contract consisting of a *Precondition* and a *Postcondition*, which are first-order predicates over the execution context and the application state. The intended semantics is as follows: When a function is called in a context that does not satisfy the precondition, the function reverts and no state change occurs. When a function is called in a state and context that satisfies its precondition, it terminates in a state which satisfies the postcondition[4].

[4] Note how this differs from preconditions e.g. in JML [6] or ACSL [1], where a function contract does not specify the behavior if the precondition is not fulfilled. These semantics would not make sense in the open, adversarial smart contracts setting.

For a given smart contract application a, we say that $\mathcal{F}$ is the set of all functions of all contracts in a, $\mathcal{V}$ the set of all state variables (qualified with the name of the contract where they are defined), and *Vals* the set of all possible values of the variables. Then the state $\mathcal{S} : \mathcal{V} \to \mathit{Vals}$ is a function which assigns each state variable a value.

For a function $f \in \mathcal{F}$, P_f is the set of all possible concrete parameter lists for f. Furthermore, we say that $\mathcal{A}$ is the set of all accounts, and the set $\mathit{Ctx} = \mathcal{A} \times P_f \times \mathbb{N}$ is the set of all call contexts. A call context $c \in \mathit{Ctx}$ is a triple consisting of the calling account, parameters, and amount of currency transferred. We write $c.\mathit{params}$, $c.\mathit{caller}$, and $c.\mathit{amt}$, respectively.

Function $\mathit{pre}_f : \mathcal{S} \times \mathit{Ctx} \to \mathbb{B}$ is the precondition of f, a predicate over the application state, the caller, and the parameters. Likewise, $\mathit{post}_f : \mathcal{S} \times \mathcal{S} \times \mathit{Ctx} \to \mathbb{B}$ is a predicate over the state before and after the execution of a function, as well as the caller and the parameters of the call.

3.2 Specification Language

In this section, we develop a specification language, based on a subset of Linear Temporal Logic, which captures our main observation about smart contract liveness properties: That is, in the context of smart contract applications, it does not make sense to specify that something "will eventually happen". Rather, one should specify that a desired functionality or state change is enabled.

Abstractly, the execution of a smart contract application can be viewed as a trace of transactions, each initiated by an account calling a function with some parameters in a way that fulfills the function's precondition. Our language enables developers to write down properties that are expected to be true in all possible execution traces.

3.2.1 LTL

Linear temporal logic (LTL, first introduced in 1977 by Pnueli [13]) is a widely used logic for describing and verifying properties of execution traces. In its original form, LTL formulas consist of a set of propositional variables, the standard boolean connectors, and some temporal operators that enable statements about traces.

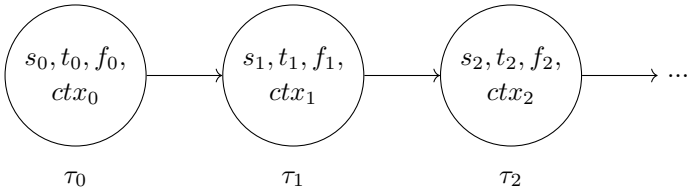
The *Next* operator $\mathbf{X}\phi$ states that some formula ϕ holds in the next step of the trace. The *Until* operator $\phi \,\mathbf{U}\, \psi$ states that eventually, ψ will hold, and ϕ must hold in every step until that point. $\Diamond$ ("eventually") with the meaning $\Diamond\phi ::= \mathit{true}\,\mathbf{U}\,\phi$ and $\Box$("globally", with $\Box\phi ::= \neg\Diamond\neg\phi$) are commonly used derivations. Furthermore, the *Weak Until* operator $\phi \,\mathbf{U_W}\, \psi$ states that ϕ must be true until a state is reached where ψ holds, but unlike $\mathbf{U}$, ψ does not necessarily have to become true at some point. The *Releases* operator $\mathbf{R}$ is the dual of $\mathbf{U}$ with $\phi \,\mathbf{R}\, \psi ::= \neg(\neg\phi \,\mathbf{U}\, \neg\psi)$. Note that both $\mathbf{U}$ and $\mathbf{R}$, in combination with negation, form a basis for the other LTL operators.

Every LTL formula can be transformed into Negation Normal Form (NNF), where the only operators are $\mathbf{U}$, $\mathbf{R}$, and $\mathbf{X}$, and where only atomic formulas are negated [15].

3.2.2 Restrictions

We adapt LTL as follows: Instead of atomic propositions, we allow predicates in first-order logic with arithmetic over the state of an application. This includes quantification over arrays and mappings, as well as over sets of accounts.

Figure 1 Our trace model of a single smart contract application execution. Each node τ_i contains the application state (s_i), the system time t_i when this state was reached, and the transaction that led to it (with each transaction consisting of the called function f_i, and the call context ctx_i).



As for the modal operators, we restrict our language in two ways: First, we omit the *Next* operator, which expresses that some condition holds in the following state. In the context of a smart contract network, it is never possible to predict which transaction is going to be executed next, as no single entity has control over this. Therefore, properties which require that something must happen exactly in the next state (as opposed to some other time) do not make sense in this domain.

Furthermore, we allow only formulas which are *Until*-free in NNF. This fragment of LTL has been called Safety LTL ([15]). Intuitively, every formula in Safety LTL rules out traces based on some finite prefix which violates the formula. This syntactic restriction captures our intuition that for smart contracts, "classical" liveness conditions (as expressed by the $\Diamond$ and $\mathbf{U}$ operators) do not make sense.

3.2.3 Domain-specific Constructs

For this restricted form of LTL, we now introduce some specification constructs which capture important "liveness" properties relevant in the smart contract domain.

We view an execution of a smart contract application as a trace (cf. Figure 1). For our purposes, each node τ_i of a trace τ consists of the application state s_i as well as the transaction which led to τ_i and the time of the transaction t_i. The transaction description consists of the name of the called function as well as the call context (caller, parameters, and transferred amount) with which it was called (we write f_i and ctx_i, respectively).

We define enabledness as a specification construct that is evaluated in one state of an execution. For a function f (with precondition pre_f over the state and the execution context) and an account a, we define that `enabled[a, par, amt](f)` is true in a node τ_i iff a can call f successfully with parameters par and amount amt in the state represented by that node:

$$\tau_i \models \texttt{enabled[a, par, amt](f)} \quad :\Leftrightarrow \quad pre_f(s_i, (a, par, amt))$$

The context, or parts of it, can be left out to indicate universal quantification, i.e., enabledness for all callers regardless of the parameters and the amount transferred with the call:

$$\tau_i \models \texttt{enabled[](f)} \quad :\Leftrightarrow \quad \forall a \in \mathcal{A}\ \forall par \in P_f : pre_f(s_i, (a, par, amt))$$

We extend this notion to predicates over the state: For a predicate over the application state p, we say that p is enabled in node τ_i if there exists a function that is enabled, and which results in a state that implies the desired state:

$$\begin{aligned}\tau_i &\models \texttt{enabled[a, par,amt](c)} \\ &:\Leftrightarrow \quad \exists f \in \mathcal{F} : \tau_i \models \texttt{enabled[a, par, amt](f)} \land post_f(s_{i-1}, s_i, (a, par, amt)) \rightarrow c\end{aligned}$$

This construct includes two-state predicates, i.e., predicates which relate two states of the application by expressing a condition over the new state in terms of the previous state. Since function postconditions can also reference the state before the function was executed, the semantics are exactly the same as for enabledness of one-state predicates.

The transfer of currency from one account to another can be described in terms of a two-state predicate. This is so predominant in the smart contract domain that we create a special abbreviation.

For accounts *from* and *to*, `transfer(from, to, amt)` is true in node τ_i iff the balances of *from* and *to* changed accordingly in the step from τ_{i-1} to τ_i:

$$\begin{aligned}\tau_i \models \texttt{transfer(from, to, amt)} \quad :\Leftrightarrow \\ s_i(from.balance) = s_{i-1}(from.balance) - amt \\ \wedge \quad s_i(to.balance) = s_{i-1}(to.balance) + amt\end{aligned}$$

Since liveness properties in smart contract applications are often about resource ownership, we introduce a special construct to express that an account a owns some amount of currency. We formalize ownership as the ability of an account to effect a transfer of currency to itself from the contract where the property is specified (`this`):

$$\tau_i \models \texttt{owns(a, amt)} \quad :\Leftrightarrow \quad \tau_i \models \Box\texttt{enabled[a]()transfer(this, a, amt)}$$

This property only makes sense when the `amt` expression refers to a variable which stores the amount, and which is updated in case of a transfer. One example is the mapping storing the balances in the Bank contract. We think this pattern is prevalent enough to justify the `owns` abbreviation.

Furthermore, we introduce a way to express that a transaction happened in a given step τ_i:

$$\tau_i \models \texttt{f[ctx]} \quad :\Leftrightarrow \quad \texttt{f} = f_i \wedge \texttt{ctx} = ctx_i$$

As with `enabled[]()`, the calling account and the parameters can be left out: `tx[]` is true in τ_i iff `tx` $= f_i$. This is useful for example when specifying that a certain condition always holds after some function was called.

Lastly, we provide a construct which describes that something (e.g., a transaction or state change) is always possible at least until it actually happens:

$$\tau_i \models \texttt{enabledUntil[ctx](f)} \quad :\Leftrightarrow \quad \tau_i \models \texttt{enabled[ctx](f)}\ \mathbf{U_W}\ \texttt{f[ctx]}$$

As above, the calling account and the parameters can be left out to indicate universal quantification, and the construct can also be used with a predicate instead of a transaction.

Thus, if p is a predicate over the application state, $f \in \mathcal{F}$ a function of the application, $par \in P_f$ a list of parameters for f, amt an integer expression, and $a, b \in \mathcal{A}$ accounts, the syntax of our language is as follows:

$$\begin{aligned}\phi ::= \ & \mathit{true} \mid \mathit{false} \\ & \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \rightarrow \phi_2 \\ & \mid \phi_1\ \mathbf{R}\ \phi_2 \mid \phi_1\ \mathbf{U_W}\ \phi_2 \mid \Box\phi \\ & \mid \texttt{enabled[}a?\ par?\ amt?\texttt{](}[p \mid f]\texttt{)} \\ & \mid \texttt{enabledU[}a?\ par?\ amt?\texttt{](}[p \mid f]\texttt{)} \\ & \mid \texttt{transfer(}a\texttt{,}b\texttt{,}amt\texttt{)} \\ & \mid \texttt{f[}a?\ par?\texttt{]} \\ & \mid \texttt{owns(}a, amt\texttt{)}\end{aligned}$$

Note that for the `enabled` and `enabledU` expressions, the account and the parameter list are optional, and the argument can be either a transaction or a predicate.

Our choice of LTL operators hints at our intentions to only care about Safety LTL formulas. As described above, we only admit formulas which are *Until*-free in NNF.

3.3 Verification

While the main focus of this work is on specification, we also want to point out that viewing smart contract liveness as a matter of enabledness can simplify the verification task.

For each property specified in an application model, the verification goal is to show that all possible executions of this application fulfill the property. For real liveness properties, as expressed by the $\Diamond$ operator, this requires additional assumptions, e.g., about the behavior of the actors in an applications. Because we restrict our specification language to safety formulas, we can, in principle, prove them by showing that the desired property is an inductive invariant.

The enabledness of a function in some context is equivalent to whether its precondition is fulfilled by that context. Likewise, the enabledness of a state or state change is equivalent to whether an enabled function exists that leads to the desired state or state change. To show that a function is always enabled, it is sufficient to show that it is enabled in the application's initial state, and that every possible call again results in a state where the function is enabled. By extension, to show that a state or state change is always enabled, it suffices to show that a function which leads to the desired state or state change is always enabled.

4 Prototypical Implementation and Evaluation

We have implemented a metamodel of smart contract applications which conforms to the model assumed for this work (cf. Section 3.1). The metamodel can be used by developers to model an application, and specify and verify the kind of liveness properties introduced in this work on the model level. Verification of application liveness properties is based on the function contracts.

Then, code skeletons consisting of function headers and function contracts (pre- and postconditions) can be generated for a specific smart contract platform (and a verification tool for this platform). After the developer completes the implementation in a way that each function conforms to its contract, the application automatically also conforms to the liveness specification.

While this model-driven approach is not the main topic of this work, we use it for a light-weight evaluation of our approach.

4.1 Implementation of Model-driven Approach

We developed a small XML-like language to describe smart contract applications consisting of several contracts. Each contract, in turn, consists of a set of state variables and functions. Our type system is inspired by Solidity, which we envision to be the main target language for our approach, but it is flexible enough to accommodate other languages as well. The type system comprises the primitive types Account, Integer (signed and unsigned), Boolean, and String. Furthermore, there are arrays and key-value mappings. Lastly, there are also the user-defined Struct and Enum types.

Each function defined by the developer has a name, a list of typed parameters, and a return type. Furthermore, we also developed a language for writing function contracts, consisting of a precondition, a postcondition, and a frame condition, which specifies which part of the application state a function may modify.

From a model written in this language, we can translate into smart contract programming languages. For evaluation, we translate into Solidity. Translating the contracts, state variables and function headers is straightforward, because the structure of our metamodel fits the structure of a Solidity application, and all of our types have an equivalent in Solidity.

The function contracts are translated to the specification language of SOLC-VERIFY[4], a tool for deductive functional verification of Solidity smart contracts. The developer provides an implementation of the generated function headers. If SOLC-VERIFYis able to prove the implementation correct against the generated formal specification, we can transfer this result back to the model, and reasonably assume that any property we can verify based on the model's function contracts is also true of the implementation.

The verification of liveness properties is not yet implemented in an automated fashion. We have developed a translation from the function contracts to an SMT encoding. For simple examples, our prototypical approach suffices to prove that a function is enabled in a given state (e.g., after initialization or after a given transaction), and that it will remain enabled, by proving that its precondition is an invariant.

4.2 Evaluation

In this section, we describe how to specify the liveness properties of the examples discussed in Section 2. We also sketch how the properties can be verified and discuss the limitations and advantages of our approach in general.

4.2.1 Bank

The main correctness property of the simple bank application (cf. Section 2.1) is that every customer can withdraw all their funds whenever they want. The customer balances are stored in a key-value mapping `bals`.

$\forall a \in \mathcal{A} : \Box$`enabled[a](a.balance ==  old{a.balance} + old{bals[a]})`

In this case, we can also specify where the money comes from, and use the `transfer` shorthand:

$\forall a \in \mathcal{A} : \Box$`enabled[a](transfer(this, a, bals[a])`

For this, we can also use the `owns` abbreviation and simply write

$\forall a \in \mathcal{A} :$ `owns(a, balances[a])`

Verification is straightforward: The `withdraw` function does not have a precondition. It is therefore always enabled for every caller, and the postcondition matches the desired property exactly.

4.2.2 Escrow

In the escrow example (Section 2.2), one liveness property is that the seller can get their deposit back after the buyer confirms the reception of the item. In our approach, this can be modeled as follows:

`confirmPurchase[buyer]` $\rightarrow$ `enabledUntil[seller](transfer(this, seller, deposit))`

This means that after the `confirmPurchase` method is called successfully, the seller is able to effect a refund of their deposit - of course, only until this actually happens.

This can be verified by showing that the only function that is enabled after the successful call to `confirmPurchase` is `refundSeller`, and that its postcondition implies the desired effect.

4.2.3 Auction

For the auction example (Section 2.3), we consider two properties: Bidders must be refunded if they do not win, and the seller should be able to claim the winning bid after the auction closes.

For the losing bidders, the property is similar to that of the bank, with the difference being that the current highest bidder can not withdraw:

$$\forall a \in \mathcal{A} : \square(\texttt{a == highestBidder} \lor \texttt{enabled[a](transfer(this, a, bals[a]))})$$

Note that in this example, we cannot use the `owns` shortcut, because it includes a $\square$ operator, so that the resulting property might actually not be true: After all, a losing bidder might increase their bid to become the highest bidder again.

Verification is also similar to the bank example. The `withdraw` function has only one precondition, which is that the caller must not be the current highest bidder. Therefore, it is always enabled for all other accounts. From this, it follows that the desired property is indeed an invariant.

For the seller in the auction, the desired property is that after the auction is closed, they get paid. This can be specified in two steps. First, after the auction ends, it can be closed:

$$\texttt{time > endTime} \rightarrow \square\texttt{enabled[](close)}$$

Second, after the auction is closed, the seller can call the `claim` function to be paid the auction price from the contract:

$$\texttt{close[]} \rightarrow \texttt{enabledUntil[seller](claim())}$$

Another correct formalization of this second property can be expressed with the application state instead of a transaction expression:

$$\texttt{state == closed} \rightarrow \texttt{enabledUntil[seller](claim())}$$

Other possible formalizations could express the ability to effect a transfer, instead of the enabledness of the claim function. In this example, there are many different reasonable ways of specifying the desired property.

Verification relies on the fact that after the `close()` function is successfully executed, the `state` variable is in the state `closed` (as implied by the postcondition). This means that the `claim()` function is enabled for the seller. Since no other function is enabled, we derive that the enabledness of the `claim()` function is an invariant until it is actually called.

At first glance, it seems that the seller property it would be easier to specify with a $\Diamond$ operator, like this:

$$\Diamond\texttt{enabled[seller](claim)}$$

This property holds, but additional assumptions about the seller's behavior would have to be given in order to be able to prove this.

4.2.4 General Remarks

Our specification language can be used to express all properties yielded by our literature research on smart contract liveness properties. This shows that the restrictions in our language (no *Next*, and only *Until*-free formulas) are actually not needed to specify properties which are commonly perceived to be liveness properties.

In some cases, our restrictions force the specification to be explicit about how a desirable state can be reached: e.g., in the auction, the specification cannot just state $\Diamond$`enabled[](close)`, but has to show the way: `time > end` $\rightarrow$ `enabledUntil[](close)`. This not only simplifies verification, but also forces clarity in the specification. If a complex sequence of function calls is necessary to reach some goal, this might point to an overly complex implementation and possible simplification. At the very least, our approach will force the developer to document the necessary steps.

There are plausible scenarios where our notion of liveness fails to express all relevant properties. One example would be a vote with a quorum: Some desirable action will be taken according to a vote, but only after a fixed percentage of those entitled to vote have cast their vote. Will the action be taken eventually? Whether or not the participants are incentivized to vote depends on the specifics of the application. If they are sufficiently incentivized, this would constitute a case where a fairness condition makes sense, and our simpler notion would not be sufficient to specify and verify that any action will be taken. However, cases like this do not seem to be common in the smart contract world, and deciding whether a fairness assumption is plausible can be very challenging. We leave this kind of question to future research.

Our model-driven approach for specification and verification enables developers to specify liveness properties on a level where the implementation of the functions is abstracted via function contracts. Therefore, we cannot rely on the implementation itself for verification. Working on the abstraction means that, in general, the properties that can be proven in our approach are a subset of the properties that would be provable directly on the implementation. However, since verification in our approach is straightforward for all example liveness properties we could find in the literature, we argue that this limitation hopefully does not matter much in practice.

5 Conclusion and Future Work

In this paper, we analyze the concept of liveness properties for smart contract applications. We find that all properties commonly perceived as liveness in the literature are not classical liveness, but can be expressed as an actor's access to some functionality. Based on this finding, we suggest a specification language based on a subset of LTL, which contains concise constructs for specifying typical properties. We also sketch how this perspective simplifies the verification task, and evaluate our approach on some typical examples.

In the future, we will look to automate the verification task. Furthermore, we will develop processes for different platforms to achieve implementations which adhere to the liveness properties specified in the model. Conversely it would be possible to translate an annotated smart contract implementation to a model, and use our approach to specify and verify liveness properties on it.

References

1 Patrick Baudin, Vincent Prevosto, Jean-Christophe Filliâtre, Yannick Moy, Benjamin Monate, and Claude Marché. ANSI/ISO C Specification Language (version 1.4), 2008.

2 Samvid Dharanikota, Suvam Mukherjee, Chandrika Bhardwaj, Aseem Rastogi, and Akash Lal. Celestial: A Smart Contracts Verification Framework. In *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 133–142, oct 2021. doi:10.34727/2021/isbn.978-3-85448-046-4_22.

3 Javier Godoy, Juan Pablo Galeotti, Diego Garbervetsky, and Sebastian Uchitel. Predicate abstractions for smart contract validation. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, MODELS '22, pages 289–299, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3550355.3552462.

4 Akos Hajdu and Dejan Jovanovic. Solc-verify: A modular verifier for solidity smart contracts. In Supratik Chakraborty and Jorge A. Navas, editors, *Verified Software. Theories, Tools, and Experiments*, pages 161–179, Cham, 2020. doi:10.1007/978-3-030-41600-3_11.

5 Daojing He, Rui Wu, Xinji Li, Sammy Chan, and Mohsen Guizani. Detection of Vulnerabilities of Blockchain Smart Contracts. *IEEE Internet of Things Journal*, pages 1–1, 2023. doi:10.1109/JIOT.2023.3241544.

6 Gary T Leavens, Albert L Baker, and Clyde Ruby. JML: A Java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, pages 404–420, 1998.

7 Matteo Marescotti, Rodrigo Otoni, Leonardo Alt, Patrick Eugster, Antti E. J. Hyvärinen, and Natasha Sharygina. Accurate Smart Contract Verification Through Direct Modelling. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, LNCS, pages 178–194, Cham, 2020. doi:10.1007/978-3-030-61467-6_12.

8 Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security*, pages 446–465, Cham, 2019. doi:10.1007/978-3-030-32101-7_27.

9 Sundas Munir and Walid Taha. Pre-deployment Analysis of Smart Contracts – A Survey, jan 2023. doi:10.48550/arXiv.2301.06079.

10 Wonhong Nam and Hyunyoung Kil. Formal verification of blockchain smart contracts via atl model checking. *IEEE Access*, 10:8151–8162, 2022. doi:10.1109/ACCESS.2022.3143145.

11 Keerthi Nelaturu, Anastasia Mavridou, Andreas Veneris, and Aron Laszka. Verified Development and Deployment of Multiple Interacting Smart Contracts with VeriSolid. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9, may 2020. doi:10.1109/ICBC48266.2020.9169428.

12 Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1661–1677, may 2020. doi:10.1109/SP40000.2020.00024.

13 Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. ieee, 1977. doi:10.1109/SFCS.1977.32.

14 Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 555–571, may 2021. doi:10.1109/SP40001.2021.00085.

15 Shufang Zhu, Lucas M Tabajara, Jianwen Li, Geguang Pu, and Moshe Y Vardi. A symbolic approach to safety ltl synthesis. In *Hardware and Software: Verification and Testing: 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings 13*, pages 147–162. Springer, 2017. doi:10.1007/978-3-319-70389-3_10.