Modelling a Blockchain for Smart Contract Verification using DeepSEA

Daniel Britten
db130@students.waikato.ac.nz
The University of Waikato
Hamilton, New Zealand

Steve Reeves
stever@waikato.ac.nz
The University of Waikato
Hamilton, New Zealand

Abstract

To create trustworthy programs, the 'gold standard' is specifications at a high-enough level to clearly correspond to the informal specifications, and also a refinement proof linking these high-level specifications down to, in our case, executable bytecode. The DeepSEA system demonstrates how this can be done, in the context of smart contracts on the Ethereum blockchain. A key component of this is the model of the blockchain on which the smart contracts reside. When doing proofs in DeepSEA, it is critical to have such a model, which allows for the writing of specifications at a high-level clearly corresponding to informal specifications. A candidate model for doing so and its usefulness for carrying out proofs is discussed in this paper.

CCS Concepts: • **Security and privacy** → **Logic and verification**; • **Computer systems organization** → *Distributed architectures.*

Keywords: smart contracts, formal methods, blockchain.

ACM Reference Format:
Daniel Britten and Steve Reeves. 2022. Modelling a Blockchain for Smart Contract Verification using DeepSEA. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS '22), December 07, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3563822.3568011

1 Introduction

In all software projects one intention is to write correct software, where "correct software" means "software that satisfies its specification". Some might say no approach exists that can produce correct software. However, we believe such an approach does exist, and this paper discusses one component in a system which demonstrates this – the DeepSEA (Deep Simulation of Executable Abstractions) system.

1.1 DeepSEA

The DeepSEA system makes it possible to write software where it is known with certainty that it is indeed correct (with respect to a formal specification) and that this has been shown with the utmost rigour. The DeepSEA compiler is partly based upon the CompCert verified compiler [5]. The CompCert verified compiler is a compiler for almost all of the C language and generates code for ARM, PowerPC, RISC-V and x86 processors. In contrast, DeepSEA as discussed here is a compiler for the DeepSEA language and generates bytecode for the Ethereum Virtual Machine (EVM).

1.2 Smart Contracts

On most blockchains, smart contracts cannot typically be updated once deployed. This constraint has the benefit that if one trusts a particular smart contract then that trust doesn't need to be reviewed again and again as the smart contract cannot change. Smart contracts often handle large sums of money or tokens representing power, so this makes it especially important that we deploy smart contracts that are guaranteed to be correct. Ideally the utmost rigour should be used in all circumstances.

1.3 Motivation for the Blockchain Model

A critical component of taking a rigorous approach is having an adequate model of a blockchain, and this is the focus of this paper. The model should be sufficiently expressive, but also abstract, to make the phrasing of lemmas related to the behaviour of smart contracts straightforward.

Ideally, any high-level model of a blockchain should be linked to a low-level model of that blockchain, such as Hirai's Lem formalization of the EVM [4]. The low-level model should be shown to *refine* the high-level model. Lem [6] is a system with a source language that has been designed to be translated into a range of programming languages and proof assistants, in particular, one target language is Coq. Hirai's Lem model has been tested against a standard test suite for Ethereum implementations which gives further assurance that the model corresponds to the behaviour of the real-world Ethereum network.

FTSCS '22, December 07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9907-4/22/12…$15.00
https://doi.org/10.1145/3563822.3568011

In this paper we focus on the version of the DeepSEA system that targets the Ethereum blockchain, and discuss a model that was developed in order to help minimise the required proof effort without compromising on correctness.

2 Refinement

The formal method *refinement* [2] formalises the informal idea of what Wirth [9] called *stepwise refinement*, a useful software engineering idea which centred around moving from more abstract programs to more concrete ones by a series of "clearly" correct steps which preserve the meaning of the program as we move through levels of abstraction.

The motivation for this is that more abstract programs are closer to an ideal statement of a solution (*e.g.* we might talk about graphs if our problem is to do with networks, even though graphs do not exist as a data type in most programming languages) and then refine to a representation of graphs in our target programming language. By solving the problem at the abstract level of graphs we are less likely to make errors in our solution since we will be writing the solution at the level of abstraction most suited to it.

The formal method uses the idea of *simulation* to connect the abstract and concrete levels at each step. A concrete program correctly simulates an abstract program as follows: if the abstract program A starts in a state s_A and moves to a final state f_A then the concrete program C, when started from any state s_C which is in a relation R with s_A, moves to some final state f_C which is in the same relation R to f_A.

Put informally, if we want a program to act like A, then we will accept as adequate a program that acts like C.

Here the relation R, which allows us to simulate A via C, relates abstract states to more concrete ones, but ensures that any properties that A has will also be properties of C. In that sense C will meet A thought of as a specification, and typically is computable even if A is not, so the refinement allows us to move, perhaps, from abstract non-computable to concrete computable.

An example of a simulation is a state that talks about sets being related to, that is simulated by, a state that talks, instead, in terms of lists. Lists are more concrete than sets, and then a further simulation of lists by arrays (in most languages, indeed, sets are not a data type, but lists might be, and arrays almost certainly are). Also, a set takes no note of ordering of its elements, whereas a list or an array does, so a single set maps to many arrays, and hence a single abstract state is simulated by many concrete states and refinement typically removes this sort of non-determinism around state as we move from abstract to concrete.

If the refinement is possible we end up with a computable program that satisfies the specification, as we might think of the abstract program being.

We note that refinement down to executable EVM bytecode is possible in principle with the blockchain model discussed in this paper, and a proof of this is left for future work. However, DeepSEA does already generate an all but completed refinement proof for each individual function of a smart contract written in DeepSEA, which forms a key part of this blockchain model.

3 The Snapshot Approach

3.1 Motivation

The focus of this model of the blockchain is on the aspects relevant to the correctness of the smart contract written in the DeepSEA language. The model's initial state corresponds with the moment when the smart contract of interest is deployed on the blockchain. The approach taken is a *snapshot* one. The model begins from an arbitrary state of the blockchain and models possible actions from then on. This helps guarantee the validity of the lemmas proven, regardless of the actual state of the blockchain.

3.2 Overview

We implement the snapshot approach by universally quantifying over all possible states the blockchain could be in at the point that the smart contract in question is deployed. This is done using a mechanism in Coq [8] called *section variables*, so that the arbitrary variables can be referred to throughout multiple definitions within the section.

This approach is used to define the reachability predicate, *ReachableFromBy*, which captures the notion of which states are reachable from an original state by a list of states and actions leading from the original state to the state in question. *ReachableFromBy* is used in the crowdfunding *donation_preserved* lemma shown in Listing 1. Starting from an arbitrary initial state in the proof helps us be sure that the result applies to the real-world Ethereum blockchain.

```
Definition since_as_long (P : BlockchainState ->
    Prop (Q : BlockchainState -> Prop) (R : Step
    -> Prop) :=
  forall sc st step',
    ReachableFromBy st step' sc ->
    P st ->
    (forall sa, List.In sa sc -> R sa) ->
    Q (Step_state step').
Notation "Q `since` P `as-long-as` R" :=
  (since_as_long P Q R) (at level 1).
Definition donation_recorded (a : addr) (amount :
    Z) (s : BlockchainState) :=
  Int256Tree.get_default 0 a (Crowdfunding_backers
      (contract_state s)) = amount
  /\ amount > 0.
Definition no_claims_from (a : addr) (s : Step) :=
  match Step_action s with
    | (call_Crowdfunding_claim _ a _ _ _) => False
    | _ => True
```

```
  end.
Lemma donation_preserved :
  forall (a : addr) (d : Z),
                (donation_recorded a d)
    `since`     (donation_recorded a d)
    `as-long-as` (no_claims_from a).
```

Listing 1. Statement of the *donation_preserved* lemma

3.3 Implementation

To implement[1] the snapshot approach Coq's section variables are used, as mentioned earlier. As shown in Listing 2, the timestamp, block number, block hashes and balances are taken to be section variables and in the context of the proofs their actual values are arbitrary.

```
Context
  (snapshot_timestamp : int256)
  (snapshot_number : int256)
  (snapshot_blockhash : int256 -> int256)
  (snapshot_balances : addr -> wei).
```

Listing 2. Section variables for the model

The *wei* datatype is *int256* which in DeepSEA is an unsigned 256-bit integer and so this guarantees the balances are non-negative.

It is possible to add additional assumptions to be used in the model such as the *address_accepts_funds_assumption* in Listing 3. Here we choose to assume that funds sent by the smart contract are always accepted by the recipient. This assumption is then used in the definition of *Action* (shown later in Listing 5) by being passed to *make_machine_env* and then used in the function responsible for checking that a transfer is successful.

```
Context (address_accepts_funds : option
    ContractState -> addr -> addr -> wei -> bool).

Definition
  address_accepts_funds_assumed_for_from_contract
    d sender recipient amount :=
      if sender =? contract_address then true
      else address_accepts_funds d sender
          recipient amount.

Definition address_accepts_funds_assumption :=
  address_accepts_funds_assumed_for_from_contract.
```

Listing 3. Example of an assumption

Based on the snapshot information we define the *initial_state* of the blockchain as in our model (which is not to be confused with the actual initial state of the blockchain back at the genesis block). This is defined as shown in Listing 4 and contains all the information assumed about the blockchain up to the point where the snapshot is taken.

[1] https://github.com/Coda-Coda/Crowdfunding/tree/FTSCS-2022. See the README for the relevant files.

```
Definition initial_state :=
  mkBlockchainState
    snapshot_timestamp snapshot_number
    snapshot_balances snapshot_blockhash
    init_global_abstract_data.
```

Listing 4. Initial state of the model

The blockchain state recorded is focused on what is relevant to the smart contract that is being verified. From this perspective, there is no need to keep track of all of the blockchain's data, only what is in Listing 4. The *init_global_abstract_data* refers to the initial values of the storage variables of the smart contract.

4 The Successful-Calls Approach

4.1 Motivation

When calling a smart contract function it may 'revert' (i.e. cancel) the current execution, which results in it returning its state to as it was before the call. Here are some examples of what may cause this.

- A runtime error in a smart contract call, such as an array out of bounds exception;
- Execution reaching a point in the smart contract code explicitly causing a revert. In DeepSEA this is done via *fail* or *assert(false)*;
- The gas cost of the transaction exceeding the gas that is provided by the sender. Modelling this scenario in DeepSEA is left for future work.

When a revert occurs, a transaction fee (gas fee) is still charged to the caller as a charge for the partial computation that was carried out. The current model implemented in DeepSEA ignores the transaction fee (and though fully accounting for gas is feasible with DeepSEA, this is left for future work). Ignoring the transaction fee means that a revert results in no change at all to the modelled blockchain state.

4.2 Overview

Typically, in a proof, the cases of interest are those where the smart contract function call succeeds. However, it is still important to model the scenario where the functions revert. From a snapshot, we can model all possible further actions. That is, we can consider the effect of all possible calls to the smart contract and then model the outcome of those calls. This is a satisfactory approach, but we can do better. Bearing in mind that the successful calls are typically the most relevant to consider, we can take a more elegant approach with our model as we are using a proof assistant. We require a proof that each call will not result in a revert and then handle the revert case separately just once. This novel approach allows us to focus the proof effort on the successful calls, while still considering all scenarios.

4.3 Implementation

The purpose of implementing[2] the successful-calls approach is to make the proofs shorter and more elegant by removing the need to repeatedly prove the statements of lemmas for the case where the state reverts over and over again.

Listing 5 defines the transitions (in a state-machine analogy) that can happen. There is one transition for each smart contract function, a transition for balance transfers, a transition for time passing, and a transition for revert.

The subtlety is that the transitions for each smart contract function require not only the arguments to the smart contract function but also a proof that, with those arguments and the current blockchain and contract state, the call to that smart contract succeeds. This is shown in the step function in Listing 6 with the inclusion of *case_donate_prf* being a required argument, and it is similar for the other smart contract functions (though hidden by the ellipsis). *case_donate_prf* guarantees that the modelled function call will not revert.

This highlights one strength of using this approach in Coq rather than relying on *tests* of the smart contract: when testing, we would need to set up the system so that we are sure, *without even running the test*, that a particular test will not cause a revert, but that is impossible in general.

```
Inductive Action (before : BlockchainState) :=
  | call_Crowdfunding_donate (context :
      CallContext)
      (callvalue_prf :
          noOverflowOrUnderflowInTransfer (caller
          context) contract_address (callvalue
          context) (balance before) = true)
      r (* The return value of calling donate
          successfully *)
      contract_state_after (* The contract state
          after calling donate successfully *)
      (case_donate_prf :
          runStateT (Crowdfunding_donate_opt (
              make_machine_env contract_address
              before context
              address_accepts_funds_assumption)) (
              contract_state before)
          = Some (r, contract_state_after))
  | call_Crowdfunding_getFunds ...
  | call_Crowdfunding_claim ...
  | externalBalanceTransfer ...
  | timePassing ...
  | revert.
```

Listing 5. Action dependent on the current blockchain state

The type of *Action* is *BlockchainState -> Type*, so it is a type which depends upon one argument of type *BlockchainState*. In particular, it depends upon the blockchain state prior to the action being executed, i.e. the argument *before*. This argument is used by *callvalue_prf* and *case_donate_prf* (or the equivalent for each function of the smart contract). The proof relating to *callvalue* ensures that the caller of the smart contract has sufficient funds to send the *callvalue* and that the smart contract's balance does not overflow on receiving those funds. The statement of this proof can only be phrased with the knowledge of the balances before the smart contract is called, hence the need for the *before* variable. The *case_donate_prf* statement relies much more heavily upon the *before* variable because *before* contains the current contract state including the value of the contract's storage variables. When 'calling' *donate* in the model, the 'caller' is required to provide a proof that *Some (r, contract_state_after)* is returned, i.e. that the contract does not revert by returning *None*. The benefit of this approach is that the various situations where the contract reverts are handled once, leaving only the cases which succeed to be proved – these typically are the cases of interest in the proof.

Consider a contract where one of its functions always reverts. For this function we would have a contradiction between the *case_function_prf* and that the function always reverts. By the principle of explosion this fulfils the proof obligation for that function's branch regardless of the specification in question. This is appropriate because the proof obligations related to when any function of the contract reverts are handled once and for all by the branch associated with the final case *revert* shown in Listing 5 and Listing 6.

The state transition *step* function for the blockchain state is defined as shown in Listing 6.

```
Fixpoint step
  (before : BlockchainState) (action : Action
      before) : BlockchainState :=
match action with
| call_Crowdfunding_donate context
    callvalue_prf r d_after case_donate_prf =>
      next_blockchain_state before d_after
| call_Crowdfunding_claim ...
| call_Crowdfunding_getFunds ...
| timePassing ...
| externalBalanceTransfer ...
| revert => before
end.
```

Listing 6. Step function

For a smart contract function, the required arguments for an action include the state of the smart contract after the call, its return value, and a proof that these are in fact the result of executing the function - with the guarantee that it does not revert. Given this, it is trivial to define the step function for smart contract function calls. The step function simply takes the resulting state of the smart contract and wraps it, after processing the possible Ether transfer to an external address which the smart contract may initiate.

As a side note, in our version of DeepSEA, a strict version of the Checks-Effects-Interactions pattern is followed as described in [1]. This means that the issue of re-entrancy

[2] https://github.com/Coda-Coda/Crowdfunding/tree/FTSCS-2022. See the README for the relevant files.

has been handled adequately and that there is at most one transfer to an external address.

5 Usage in a Crowdfunding Proof

By making use of the above approach to modelling the blockchain, we proved a lemma relating to the preservation of records of donations in a Crowdfunding smart contract. The proof is shorter than the original proof that it is based upon an approach using Scilla [7] which used a simpler system that is not compatible with a verified compiler. This original proof serves as a useful benchmark, having also been done in Coq. Our proof is structurally simpler and shorter, while still being compatible with the DeepSEA verified compiler. The full proof is shown in Listing 7 to Listing 10.

The tactics (proof commands) in Listing 7 progress the proof state to the goal being the left side of *donation_recorded* (shown in Listing 1), having dismissed by *assumption* that *d > 0*.

```
Lemma donation_preserved :
  forall (a : addr) (d : Z),
                (donation_recorded a d)
    `since`     (donation_recorded a d)
    `as-long-as` (no_claims_from a).
Proof.
(* Proof of a temporal property. *)
unfold since_as_long. intros.
induction H; [assumption|].
assert(donation_recorded a d prevSt) by (apply
    IHReachableFromBy;
intros; apply H1; apply in_cons; assumption).
clear H0 IHReachableFromBy.
unfold donation_recorded in *. destruct_and.
split; [|assumption].
```

Listing 7. *donation_preserved* proof (part 1)

The next tactics in Listing 8 split the goal into each of the possible actions that can be taken in the model of the blockchain. The trivial cases of handling external balance transfers, time passing and reverts are dismissed by *assumption*, leaving only the goals for the cases for calling the functions *donate* and *getFunds*. This is appropriate because it is only the smart contract functions directly relate to business logic of handling donations and thus are relevant for the *donation_preserved* lemma.

```
Hlinks. assert (no_claims_from a prev) by
(apply H1; destruct HL; subst; right; left; auto).
destruct prev; autounfold in *; simpl in *.
clear H1 H HL. unfold no_claims_from in H3.
unfold donation_recorded in *.
destruct Step_action0;
  simpl in *;
  rewrite <- HS in *;
  try assumption.
```

Listing 8. *donation_preserved* proof (part 2)

In Listing 9 the case for the *donate* function is proven. The two sub-cases relate to whether the arbitrarily chosen address *a* is the person currently donating or not. The lemmas *get_default_ss* and *get_default_so* capture the notions of getting the value with the same key just after it has been set "get set same" (the value returned is the one just set) and getting the value of a different key just after it has been set "get set other" (the value returned is unchanged by the set).

```
- Transparent Crowdfunding_donate_opt.
  unfold Crowdfunding_donate_opt in *. ds_inv;
      subst; simpl; inv_FT.
  destruct (a =? (caller context)) eqn:Case.
  + apply Int256eq_true in Case. rewrite <- Case.
    rewrite get_default_ss. exfalso. subst. simpl
        in *.
    apply Z.eqb_eq in Heqb0. rewrite Heqb0 in H2.
        lia.
  + apply Int256eq_false in Case. apply
      get_default_so; assumption.
```

Listing 9. *donation_preserved* proof (part 3)

The final tactics in the proof (Listing 10) solve the case of the *getFunds* and *claim* function. The function *getFunds* does not alter the donation record, it simply sets the *funded* flag to *true* to prevent later attempts to withdraw funds after the owner has withdrawn all funds. Since the donation record is unchanged in all branches of *getFunds* the tactics "*subst; reflexivity*" solve all the goals by substituting variables, and observing that the goal is of the form "*a = a*" after simplification.

The case of calling the *claim* function (for the backers to withdraw funds after the campaign fails to meet its goal) gives rise to a trivial contradiction because of the "*as-long-as*" clause in the statement of the lemma. This is evident from the definition of *no_claims_from* in Listing 1, in particular that the branch for *claim* is *False* (which is captures the notion of a contradiction in Coq).

```
- Transparent Crowdfunding_getFunds_opt.
  unfold Crowdfunding_getFunds_opt in *.
  ds_inv; subst; reflexivity.
- contradiction.
Qed.
```

Listing 10. *donation_preserved* proof (part 4)

As shown in the listings above, the structure of the proof is indicated by the bullets of - and +. As you can see, there are only three main bullets and two sub-bullets. This gives an indication that the proof itself does not require too many branches. This simplicity is a direct result of the successful-calls approach, because the trivial revert case only has to be proved once.

Listing 11 shows the the proof state[3] at the point of the first bullet. As you can see *case_donate_prf* from the *Action*

[3]Generated with the aid of Coq Proof State Visualiser [3].

Proof situation:

$$
\begin{gathered}
contract\_address,\ a : addr \\
d : Z \\
prevSt : BlockchainState \\
context : CallContext \\
\textbf{\textit{callvalue\_prf}} : noOverflowOrUnderflowInTransfer \\
(caller\ context)\ contract\_address \quad (callvalue\ context) \\
(balance\ prevSt) = true \\
r : unit \\
contract\_state\_after : global\_abstract\_data\_type \\
\textbf{\textit{case\_donate\_prf}} : runStateT \\
(Crowdfunding\_donate\_opt \quad (make\_machine\_env \\
contract\_address\ prevSt\ context)) \quad (contract\_state \\
prevSt) = Some\ (r, contract\_state\_after)
\end{gathered}
$$

$$
\begin{gathered}
get\_default\ 0\ a\ (Crowdfunding\_backers \\
contract\_state\_after) = d
\end{gathered}
$$

Listing 11. Simplified proof situation at the first bullet of *donation_preserved*

type appears in the proof context. Since *case_donate_prf* is of the form $runStateT\ (Crowdfunding\_donate\_opt\ \ldots = Some\ (r, contract\_state\_after)$ we have in the proof context that the *donate* function does not return *None* (i.e. does not revert). We also have *callvalue_prf* in the proof context, which guarantees no revert occurs as a result of insufficient or an excess of funds related to the funds transferred by the caller to the contract.

6 Discussion

A limitation of this model is that it does not (yet) handle gas constraints. It implicitly assumes that the gas cost paid by the sender is always sufficient for the transaction to complete. DeepSEA does actually track gas usage but it does not yet do so at the high-level of abstraction used in this model.

Although the purpose of the successful-calls approach is to simplify the proof effort, there are some ways in which it does make proofs slightly more complex. The complexities arise from the use of the *Action* type depending on a value of type *BlockchainState*. For example, an intuitive approach as a part of modelling a blockchain is to have a separate type for a list of states and a list of actions. However, to do this with the *Action* type dependent on a *BlockchainState* it is necessary to instead have a list of dependent records which include both a state and action in each element of the list.

An additional challenge is that of ensuring the verification system stays current. For example, there have been changes[4] to the test set which Hirai's Lem model [4] was tested on[5] but Hirai's Lem model has not been updated since 2018. Updating Hirai's Lem model and any required changes in DeepSEA would be important before the full benefits of the DeepSEA system could be realised.

7 Conclusion

This paper discusses modelling the Ethereum blockchain in Coq by making use of the ideas of a snapshot and a successful-calls technique leveraging Hirai's Lem model of the EVM and the DeepSEA system. These ideas help simplify proofs without compromising on correctness. The modelling is expressive enough that it can be used to express properties about smart contracts at a high level.

Future work would involve demonstrating that the aspects of the model which 'glue' the individual functions together into a model of the contract (and blockchain) as a whole are also refined by Hirai's low-level Lem model [4]. In the meantime, careful inspection of the aspects which 'glue' the individual functions together as a part of the high-level model is required, with them being part of the trusted computing base for now. Such careful inspection could involve examining the model's Coq source code to see if it corresponds with common knowledge about how the Ethereum blockchain functions, or perhaps comparison with the Ethereum Yellow Paper [10]. The first of these has been done so far. The high-level nature of the model lends itself to this kind of analysis. Nevertheless, future work involving a refinement proof for all aspects of the model would give greater assurance the model is correct.

Acknowledgements. I (Daniel Britten) would like to thank Vilhelm Sjöberg for many helpful discussions related to this research. I would also like to thank the University of Auckland and Associate Professor Jing Sun for hosting me during this research.

References

[1] Daniel Britten, Vilhelm Sjöberg, and Steve Reeves. 2021. Using Coq to Enforce the Checks-Effects-Interactions Pattern in DeepSEA Smart Contracts (Short Paper). In *3rd International Workshop on Formal Methods for Blockchains (FMBC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. https://doi.org/10.4230/OASIcs.FMBC.2021.3

[2] John Derrick and Eerke Boiten. 2018. *Refinement: Semantics, Languages and Applications*. Vol. 95. Springer.

[3] Mario Frank. 2021. The Coq Proof Script Visualiser (coq-psv). *arXiv preprint arXiv:2101.07761* (2021). https://doi.org/10.48550/arXiv.2101.07761

[4] Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*. Springer, 520–535. https://doi.org/10.1007/978-3-319-70278-0_33

[4] Latest version of relevant tests for Hirai's Lem model: https://github.com/ethereum/tests/tree/develop/GeneralStateTests/VMTests.

[5] Likely version of the tests Hirai's Lem model was tested on: https://github.com/ethereum/tests/tree/79b14fbd74c0148f1b8cca442252e9299464fe8c/VMTests.

[5] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert – A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. SEE, Toulouse, France. https://hal.inria.fr/hal-01238879

[6] Scott Owens, Peter Böhm, Francesco Zappa Nardelli, and Peter Sewell. 2011. Lem: A Lightweight Tool for Heavyweight Semantics. In *Interactive Theorem Proving*, Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 363–369. https://doi.org/10.1007/978-3-642-22863-6_27

[7] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. 2018. Scilla: a Smart Contract Intermediate-Level LAnguage. *arXiv:1801.00687 [cs]* (Jan. 2018). https://doi.org/10.48550/arXiv.1801.00687 arXiv: 1801.00687.

[8] The Coq Development Team. 2022. The Coq Proof Assistant. (Jan 2022). https://doi.org/10.5281/zenodo.5846982

[9] Niklaus Wirth. 1971. Program Development by Stepwise Refinement. *Commun. ACM* 14, 4 (apr 1971), 221–227. https://doi.org/10.1145/362575.362577

[10] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

Received 2022-09-08; accepted 2022-10-10