Formally Verifying a Payment Channel System

Christopher Fu
Zhong Shao
Vilhelm Sjöberg

May 3, 2017

Abstract

Ethereum is a public, open-source platform that allows users to create contracts that can store data on the Ethereum blockchain and send and receive messages to and from other contracts. Such contracts can be combined in complex ways to create powerful decentralized applications. Because of the nature of the Ethereum blockchain, bugs in contract code can be difficult to spot, hard to rectify, and lead to large monetary losses if exploited. Originally, we proposed to use DeepSEA, a certified programming language developed to allow developers to write specifications for program behavior, to write formally verified smart contracts. We model a payment channel system where parties can make off-chain transactions and then resolve disputes via an on-chain contract. Such a payment channel system is useful because it enables parties to quickly perform transactions without waiting for blocks to be mined on the blockchain while also relying on an on-chain contract to ensure that the parties get paid what they are owed. We prove the correctness of this model by showing that our client and on-chain contract specifications allow a truthful client to receive at least what he is owed, even when faced with an untruthful adversary. We were able to rewrite the on-chain contract specification in DeepSEA, but because of ongoing work on the DeepSEA frontend, we were ultimately unable to get the compiler to generate a usable Coq specification to use with our proof of correctness. Further work is necessary to integrate the work done here with existing efforts to construct a DeepSEA backend that generates Ethereum Virtual Machine bytecode.

1 Introduction to Ethereum

Ethereum is a public, open-source platform that allows users to build and run decentralized applications that run on blockchain technology. A blockchain is a distributed computing architecture where each node in the network executes and records transactions as blocks in the blockchain. Because only one block is added at a time and each block contains a cryptographic proof of its validity, each node in the network can agree upon the current state of the blockchain.

The blockchain forms the backbone of Bitcoin, a cryptocurrency released in 2009, which uses it to record transactions between users. Ethereum, on the other hand, keeps track of the state of every account on the blockchain. The account is the basic building block of Ethereum, of which there are two types: externally owned accounts (EOAs) and contract accounts (contracts). EOAs are human-controlled because they are controlled by private keys held by the accounts owner. Contracts (often also referred to as smart contracts), on the other hand, are a collection of code and data that reside at a specific address on the Ethereum blockchain in a binary format called Ethereum Virtual Machine (EVM) bytecode. Contracts, which are typically written in a higher-level language like Solidity and compiled into EVM bytecode to be deployed to the blockchain, can send and receive messages to and from each other and to accounts. In this manner, they can be put together to build decentralized applications backed by the Ethereum blockchain. Examples of such applications range from voting applications, to peer-to-peer trading markets, to video games.
2 Original project proposal

Originally, we proposed to use DeepSEA, a certified programming language developed to allow developers to write specifications for program behavior, to write formally verified smart contracts. This way, we would be able to guarantee that the behavior of such contracts will be well-defined and bug-free. DeepSEA was originally used to aid in the development of mCertiKOS, a fully verified hypervisor that can boot a version of Linux as a guest. Edsger, the DeepSEA compiler frontend, and a Coq-CompCertX backend are used to generate a Coq specification, a machine code binary, and a proof that the two are equivalent. We can then use the generated Coq specification to prove useful properties about the program.

The ultimate goal of the project was to retool DeepSEA to also be able to model smart contracts. We first looked to model the structure of some typical Solidity contracts to determine what some useful properties to prove about contracts looked like. This way, we hoped to be able to gain a better understanding of what a potential intermediate representation for the DeepSEA frontend would look like. In addition, a new backend must be written to generate EVM bytecode so that the compiled program can be run on the Ethereum blockchain.

3 Motivation

We wanted to model a system in Coq that could potentially be an Ethereum application. We decided on a payment channel system, which is an off-chain mechanism for performing transfers of credit between two parties (or players). The process of performing a transaction over a payment channel looks roughly like the following:

1. **Open the channel.** Both parties deposit some amount of credit with an on-chain transaction as a security deposit.
2. **Make off-chain transactions.** The parties can now make payments to each other. They do this by exchanging cryptographically signed messages containing the next proposed round state, which consists of a round number and balance.
3. **Trigger.** Once a party decides to finalize the transaction, it notifies the on-chain contract that the payment channel will be closed and submits the proposed payouts. The contract then notifies the other party and waits for a fixed amount of time for any disputes. If there is a dispute between the parties about the amounts paid, both parties can send evidence (e.g., the signed messages) to the on-chain contract, which then resolves the dispute based on the evidence.
4. **Finalize.** Once the time limit for disputes has passed, the on-chain contract then distributes the correct amount of ether to both parties.

The main benefit of using payment channels instead of directly performing transactions on the blockchain is that the parties do not have to wait for the block containing the transaction to be mined for the payment to be “confirmed.” Currently, the amount of time it takes for an Ethereum block to be mined is around 12 seconds, which is too slow to facilitate high-frequency transactions. Using a payment channel, two parties can exchange many off-chain transactions, thereby sidestepping the block time issue, and then take advantage of the blockchain to pay out credits and resolve disputes at the very end.

Originally, I started by looking at the code provided by the Miller 2017 paper, intending to base my Coq model on the model described in the paper. I focused on `contractPay.sol` and `test_pay.py`, which are the implementation of the on-chain contract and some test driver code, respectively. However, I found that following along with their implementation was not suitable for our purposes, primarily because their payment channel model depended on a test driver that

---

1 Coq is a formal proof assistant that provides a language to write programs and prove properties about them, and CompCertX is a variant of CompCert, a formally verified C compiler that can generate code for x86, PowerPC, and ARM processors.
required both parties to operate in a synchronous manner (i.e. both parties make transactions with each other at the same time). Therefore, we decided to write a new specification for the payment channel smart contract and the client code.

4 Proof of correctness of the player and on-chain contract specifications

4.1 Definition of correctness

Our model of the payment channel system consists of specifications of the client and smart contract code, and a description of how the two clients (or players) and the contract can interact. Ultimately, we wish to show that honest players whose actions adhere to our client specification will be able to receive at least what they are owed, even if the other player in the payment channel is dishonest.

Because our model is symmetric, without loss of generality, consider the point of view of an honest player (L for left) communicating with a possibly dishonest player (R for right). We define the correctness of our payment channel system specification as follows: no matter what actions R takes, L is able to ultimately receive at least what he is owed. What, then, is the correct amount that L should be owed? If we keep track of all signed round states made by both L and R in an append-only log, and if a new round state can only be added to the log if both players agree upon the validity of the transaction that results in that new round state, then let the latest entry in the log be the true state of the system. If either one of the players presents the true state to the on-chain contract, and it verifies that the state is valid, then the amount that each player is paid in the true state after the payment channel is finalized is defined to be the “correct” amount. Note that under our definition of correctness, L could potentially be paid more than he is owed; however, L cannot be paid less than he is owed.

| Inductive can_force (st : SystemState) : Prop := |
| cf_finished : (let !(_, cst, rd_st_sigs) := st in |
| PmtChan.left_withdrawal cst >= |
| PmtChan.left_deposit cst + match rd_st_sigs with |
| [] => 0 |
| (_ , delt , _) :: _ => delt |
| end) -> |
| can_force st |
| cf_left_step : |
| forall st', step SideT.Left st st' -> |
| (forall st', multistep_right st' st' -> can_force st') -> |
| can_force st. |

Listing 1: Definition of the can_force property.

In our proof, we call this property can_force (Listing 1)—that is, L “can force” the system towards a final state in which he is paid at least what he is owed. We say that a state is can_force if either L has already been paid at least what he is owed, or he can perform an action such that for all actions that R can take after that action, the resulting state is still can_force. If we can prove that starting from the initial state of the system, all reachable states are can_force, then we claim our system to be correct. We show this theorem in Listing 2. PmtChan.init, Player.init, PmtChan.deposit, and Player.deposit are functions that initialize the on-chain contract and left player, and make the initial deposits. multistep is an inductive relation that describes zero or more state transitions, or steps (Listing 3). We will describe the step relation more thoroughly in Section 4.2.
Theorem can_force_thm :
    forall left_addr right_addr
    left_deposit right_deposit
    pst cst cst' cst'' st',
    Some cst = PmtChan.init left_addr right_addr ->
    pst = Player.init left_addr right_addr SideT.Left ->
    Some cst' = Player.deposit pst cst left_deposit ->
    Some cst'' = PmtChan.deposit (Build_Msg right_addr right_deposit) cst' ->
    multistep (pst, cst'', []) st' ->
    can_force st'.

Listing 2: can_force_thm is the ultimate correctness theorem of the system.

Inductive multistep : SystemState -> SystemState -> Prop :=
| multistep_refl : forall st, multistep st st |
| multistep_step : forall side st st' st'', step side st st' ->
  multistep st' st'' ->
| multistep step st'''.

Listing 3: multistep represents zero or more steps.

4.2 Definition of the step relation

In order to reason about how the client and contract code interact, we define a model in Coq
with a step relation that defines how the system can change over time. Currently, this is part
of the trusted computing base. However, in the future, with more work on the DeepSEA compiler,
we should be able to reduce the amount of trust necessary, as the DeepSEA compiler will be
able to compile the contract code and generate a Coq proof of equivalence to the outputted
specification.

Our system allows the following steps to occur:

1. L pays R
2. L receives from R (i.e. R pays L)
3. L triggers the on-chain contract
4. R triggers the on-chain contract
5. L presents evidence to the on-chain contract
6. R presents evidence to the on-chain contract
7. L finalizes the payment channel

Remember that, as mentioned in Section 4.3, one of the simplifications we have made is that R
cannot finalize the payment channel. We can visualize these steps in the context of the state
machine shown in Figure 1, where the states represent the possible states that the on-chain
contract can be in.

In Coq, we describe the possible state transitions in an inductive relation named step, which
is a function of the side of the player performing the action (i.e. either L or R) and the initial
and final SystemStates, which are tuples of the PlayerState, the ContractState, and the
log of signed messages. We show the step relation in Listing 4.
Definition ContractState := PmtChan.State.
Definition PlayerState := Player.State.
Definition SystemState : Type := PlayerState * ContractState * list (RoundState * Sig).

Inductive step SideT.Side -> SystemState -> SystemState -> Prop :=
| left_pay : forall pst cst (pst' : PlayerState) rd_st_sigs rd_st_sig amt,
PmtChan.trigger msg pst cst cst' rd_st_sigs,
Player.recv pst cst (rd_st, sig) = Some pst' ->
step SideT.Right (pst, cst, rd_st_sigs)
| right_pay : forall cst (sig : Sig) (pst' : PlayerState) (cst' : ContractState)
pst' rd_st_sigs rd_st sig,
sig = sign_round_state rd_st (Player.other_addr pst) ->
Player.recv pst cst (rd_st, sig) = Some pst' ->
step SideT.Right (pst, cst, rd_st_sigs)
| left_trigger : forall pst cst pst' cst' rd_st_sigs, 
Player.trigger pst cst = Some (pst', cst') ->
step SideT.Left (pst, cst, rd_st_sigs) (pst', cst', rd_st_sigs)
| right_trigger : forall msg pst cst cst' rd_st_sigs, 
PmtChan.trigger msg cst = Some cst' ->
step SideT.Right (pst, cst, rd_st_sigs)
| left_update_pending_rd_st_sig : forall pst cst rd_st_sigs cst', 
Player.update_pending_rd_st_sig pst cst = Some cst' ->
step SideT.Left (pst, cst, rd_st_sigs) (pst', cst', rd_st_sigs)
| right_update_pending_rd_st_sig1 : forall msg pst cst rd_st_sigs new_rd_st_sig cst', 
In new_rd_st_sig rd_st_sigs ->
sender msg = PmtChan.right_player cst ->
PmtChan.update_pending_rd_st_sig msg cst new_rd_st_sig = Some cst' ->
step SideT.Right (pst, cst, rd_st_sigs) (pst', cst', rd_st_sigs)
| right_update_pending_rd_st_sig2 : forall msg pst cst cst' rd_st rd_st_sigs, 
sender msg = PmtChan.right_player cst ->
PmtChan.update_pending_rd_st_sig msg cst
| left_finalize : forall pst cst pst' cst' rd_st_sigs,
Player.finalize pst cst = Some (pst', cst') ->
step SideT.Left (pst, cst, rd_st_sigs) (pst', cst', rd_st_sigs).

Listing 4: The step relation describes all possible state transitions of the system.
Each case of the step relation consists of either a Player or PmtChan function that, depending on the function, returns an option of the new PlayerState, ContractState, or a tuple of both (and in the case of Player.pay, the signed new round state after the transaction). On success, the functions return Some: on failure, None.

For example, for the left_trigger case, which represents \( L \) triggering the on-chain contract (step 3 in Section 4.2), we pass the current player state \( pst \) and contract state \( cst \) to Player.trigger (Listing 5), which checks if the player state’s current status is \( \text{Ok} \) and then, in turn, calls PmtChan.trigger (Listing 6) to trigger the on-chain contract. If the PmtChan.trigger call is successful, then PmtChan.trigger updates the player status to Triggered and then returns the new player and contract states.

Note that there are two cases for \( R \) presenting evidence (step 6). right_update_pending_rd_st_sig1 is the case in which \( R \) presents a round state signature that either \( L \) or \( R \) has signed before, and right_update_pending_rd_st_sig2 is the case in which \( R \) forges a new round state signature signed by \( R \) himself.

Of particular importance is PmtChan.update_pending_rd_st_sig (Listing 7), which verifies the presented round state signatures when the payment channel is triggered (steps 5 and 6), and if the signature is valid, stores it as the pending round state signature in the pending_rd_st_sig field. PmtChan.update_pending_rd_st_sig only accepts round state signatures that are signed by the opposite party. For example, when \( R \) calls PmtChan.update_pending_rd_st_sig, it must present a round state signature signed by \( L \) (i.e. a transaction in which \( L \) pays \( R \)). This means that \( R \) cannot present a forged signature that has not been seen before because PmtChan.update_pending_rd_st_sig will reject signatures signed by \( R \) himself if the caller is \( R \).

In addition, the on-chain contract stores the signature with the largest round number, and the player specification only makes and accepts transactions that increase the round number, so the pending round state signature represents the latest transaction seen by the on-chain contract. This means that if \( R \) presents a signature that is too old and not reflective of the current state of the payment channel, \( L \) can present a newer state, and the on-chain contract will resolve the dispute in favor of \( L \).

---

Listing 5: Player.trigger triggers the on-chain contract and allows it to begin accepting round state signatures as evidence of transactions.

```haskell
definition trigger (st : State) (cst : PmtChan.State) : option (State × PmtChan.State) :=
  if Status.eqb st.(status) StatusT.Ok then
    match PmtChan.trigger (Build_Msg st.(addr) 0) cst with
    | None => None
    | Some cst' => Some (st {status: StatusT.Triggered}, cst')
  end
else
  None.
```

Listing 6: Definition of PmtChan.trigger. only_players verifies that the sender of msg is one of either of the two players.

```haskell
definition trigger (msg : Msg) (st : State) : option State :=
  if only_players msg st && Status.eqb st.(status) StatusT.Ok then
    Some (st {status: StatusT.Triggered})
  else
    None.
```
Listing 7: PmtChan.update_pending_rd_st_sig verifies and updates the pending round state signature stored by the on-chain contract.

4.3 Assumptions and simplifications made to the protocol

We first make note of certain assumptions and simplifications that we made to the protocol described in Section 3 in our specifications to simplify our proofs:

1. We eliminate the timeout that the on-chain contract waits for after it has been triggered and only allow L to finalize the payment channel. We cannot allow R to finalize because if R finalizes immediately after triggering, L would have no chance to present signed messages to the on-chain contract. This is a significant simplification, but it should have little effect on the overall thrust of the proof of correctness, and it should be easily rectifiable with some additional work.

2. We represent a signed message (digital signature) as a simple inductive data type. Once a message is signed, the signature can only be verified. This is using the symbolic model of interactive cryptographic protocols, where cryptographic primitives are modeled by abstract operators (cf. the Dolev-Yao model [6]).

3. Ethereum addresses are simply represented as opaque 160-bit CompCert Integers. They do not support any member accessors or methods, such as balance and send.

4. The balance element of the round states exchanged by the players and the on-chain contract is a “delta” relative to L. In other words, when the payment channel is first established, the round state delta = 0. If L pays R 10 ether, delta becomes -10. After that, if R pays L 20 ether, delta becomes 10.

4.4 Proof structure

We now have enough to describe the overall proof of correctness structure. Let the round state signature corresponding to the “true state” of the payment channel be sig_true, and its delta be delta_true.
First, let’s consider the case where R has made no payments to L, in which case L does not need to present any signatures. If R also does not present any signatures, L will be required to pay nothing. If R does present a signature, the delta of that signature is guaranteed to be in the range \( 0 < \Delta \leq \Delta_{\text{true}} \). In either case, L is guaranteed to pay at most \( \Delta_{\text{true}} \) to R, which satisfies can_force.

Next, let’s consider the cases where R has made one or more payments to L. L keeps track of the last signature signed by R, which is a payment from R to L. Let this last payment be \( \text{sig}_{\text{last}} \), and the delta of \( \text{sig}_{\text{last}} \) be \( \Delta_{\text{last}} \). We know that if L has made no subsequent payments to R after \( \text{sig}_{\text{last}} \), then \( \text{sig}_{\text{last}} = \text{sig}_{\text{true}} \). If L has made one or more subsequent payments, then \( \Delta_{\text{true}} < \Delta_{\text{last}} \) because all transactions after \( \text{sig}_{\text{last}} \) must be ones where L pays R, and thus \( \Delta \) monotonically decreases with increasing rd_num. This means that \( \Delta_{\text{true}} \) is the smallest \( \Delta \) among the \( \Delta_{\text{last}} \) of the transactions after \( \text{sig}_{\text{last}} \).

After the payment channel has been triggered by either L or R, one of following four possibilities may occur:

1. R has made no payments to L. L needs to take no additional action, and the system is already can_force.

2. R has made one or more payments to L.
   (a) The pending round state signature is newer than \( \text{sig}_{\text{last}} \). L needs to take no additional action because, as stated above, for all signatures that are newer than \( \text{sig}_{\text{last}} \), \( \Delta \) monotonically decreases with increasing rd_num, which \( \Delta_{\text{true}} \) being the smallest such \( \Delta \). Thus, no matter which signature R chooses to present, L is guaranteed to pay at most \( \Delta_{\text{true}} \), and the system is can_force.
   (b) The pending round state signature is older than \( \text{sig}_{\text{last}} \). L must present \( \text{sig}_{\text{last}} \). Using the same reasoning as in (a), L is guaranteed to pay at most \( \Delta_{\text{true}} \), and the system is can_force.
   (c) No player has presented a state yet. L must present \( \text{sig}_{\text{last}} \). The reasoning, once again, is the same as in (a).

In Coq, we consider each of these four cases in Lemmas A.1–A.4. Each of these lemmas requires the starting state \( \text{st} \) to satisfy system_invs (Listing A.5), which is a conjunction of invariants that must hold at the beginning of the protocol and after each step. can_immediately_force is an auxiliary property that implies can_force. A state is can_immediately_force if it is either already can_force, or the payment channel can be finalized and the resulting state is can_force.

One such family of system invariants that proved to be particularly important is the find_split_* invariants, which describe the properties of find_split (Listing A.8). find_split is a function similar to find that also splits the input list into two lists that contain the elements that come before and after the element satisfying the predicate \( f \). This is useful, for example, to state the invariant that in the log of round state signatures, for any given signature in the log, a signature \( \text{sig} \) has a round number greater than that of the found signature \( \text{sig}_f \) that satisfies \( f \) if and only if \( \text{sig} \) comes before \( \text{sig}_f \) in the log (new elements are cons’d to the head of the log) (Listing A.8).

Once we have proven Lemmas A.1–A.4, we know that if the system is in a Triggered state, it is can_force. From here, it is relatively straightforward to show that paying and receiving do not affect the can_force ableness of the system, and thus all reachable states from the initial state are can_force.

5 Rewriting the on-chain contract specification in DeepSEA

Once we completed the proof of the payment channel system, we looked into rewriting the contract specification in DeepSEA. DeepSEA organizes its functions into class-like objects that define methods that can access and mutate internal state. We converted our on-chain contract specification to a DeepSEA object containing fields for keeping track of deposits, the pending
Fixpoint find_split {A} (f : A -> bool) (before l : list A) :
  (list A * option A * list A) :=
match l with
  | [] => (before, None, [])
  | x :: xs => if f x then
    (before, Some x, xs)
  else
    find_split f (before ++ [x]) xs
end.

Listing 8: find_split splits the input list into two lists containing the elements before and after the found element.

Definition find_split_rd_num_gt_inv (st : SystemState) : Prop :=
  forall bef found aft f rss,
  let '(_, _, rd_st_sigs) := st in
  find_split f [] rd_st_sigs = (bef, found, aft) ->
  In rss rd_st_sigs ->
  (match found with
   | Some (rd_num, _, _) =>
     let '(rss_rd_num, _, _) := rss in
     rss_rd_num > rd_num
   | None => True
   end <->
  In rss bef).

Listing 9: find_split_rd_num_gt_inv is an invariant that makes use of find_split to describe a property about signatures that come before the found signature.

round state signature, etc. and init, deposit, withdraw, etc. methods corresponding to the state-modifying functions found in the Coq specification (Listing 1).

The resulting DeepSEA code was able to typecheck, but the frontend is currently undergoing a heavy rewrite, so there are some bugs that prevent it from generating a working Coq specification. We were ultimately unable to rewrite our proofs using the generated specifications in the time remaining.

6 Conclusion

We were able to construct a simplified payment channel system in Coq that models two clients performing off-chain transactions and a smart contract running on the Ethereum blockchain that resolves disputes between the two clients. We proved an interesting, non-trivial correctness property of the system—that our client specification is able to be paid at least what he is owed, even against an untruthful adversary. We were also able to rewrite the on-chain contract in DeepSEA, but we were ultimately unable to get the DeepSEA frontend to generate a usable Coq specification to use with our proofs.

Moving forward, there are several areas that offer promising opportunities for further exploration:

1. Add a timeout so that simplification mentioned in Section 4.3 is no longer necessary.
2. Integrate the specifications and proofs with a formalized model of Ethereum. Currently, our model of how Ethereum works is very ad-hoc.
3. Fix the DeepSEA frontend to generate usable Coq specifications, and make the proof of correctness work with the generated Coq specification from the DeepSEA specification of the on-chain contract.
object PmtChan (utils : UtilsSig) : PmtChanSig {
    let left_player : uint := 0u0
    let right_player : uint := 0u0
    let left_deposit : int := 0
    let right_deposit : int := 0
    (* many more fields omitted... *)

    let init (l_player, r_player) = (* omitted *)
    let deposit msg = (* omitted *)
    let withdraw msg = (* omitted *)
    let trigger msg = (* omitted *)
    let update_pending_rd_st_sig (msg, rd_st', sig) = (* omitted *)
    let finalize msg = (* omitted *)
}

Listing 10: The PmtChan object in DeepSEA encapsulates the same state and methods that are defined by the PmtChan module in the Coq specification.

4. Integrate the work done so far with existing efforts to write a DeepSEA backend that generates EVM bytecode.
A Appendix

A.1 Listings

Listing A.1: Lemma for case 1, when $R$ has made no payments to $L$.

```
Lemma enter_cif_after_trigger_porss_is_none : forall (st : SystemState),
let '(pst, cst, _) := st in
system_invs st ->
PmtChan.status cst = PmtChan.StatusT.Triggered ->
Player.other_rd_st_sig pst = None ->
can_immediately_force st.
```

Listing A.2: Lemma for case 2a, when the pending round state signature is newer than $\text{sig}_{last}$.

```
Lemma enter_cif_after_trigger_porss_is_some1 :
forall (st : SystemState) other_rd_st_sig pending_rd_st_sig,
let '(pst, cst, _) := st in
system_invs st ->
PmtChan.status cst = PmtChan.StatusT.Triggered ->
Player.other_rd_st_sig pst = Some other_rd_st_sig ->
PmtChan.pending_rd_st_sig cst = Some pending_rd_st_sig ->
let '(orss_rd_num, orss_delt, _) := other_rd_st_sig in
let '(prss_rd_num, prss_delt, _) := pending_rd_st_sig in
orss_rd_num <= prss_rd_num ->
can_immediately_force st.
```

Listing A.3: Lemma for case 2b, when the pending round state signature is older than $\text{sig}_{last}$.

```
Lemma enter_cif_after_trigger_porss_is_some2 :
forall st other_rd_st_sig pending_rd_st_sig,
let '(pst, cst, _) := st in
system_invs st ->
PmtChan.status cst = PmtChan.StatusT.Triggered ->
Player.other_rd_st_sig pst = Some other_rd_st_sig ->
PmtChan.pending_rd_st_sig cst = Some pending_rd_st_sig ->
let '(orss_rd_num, _, _) := other_rd_st_sig in
let '(prss_rd_num, _, _) := pending_rd_st_sig in
orss_rd_num > prss_rd_num ->
eexists st',
    step SideT.Left st st' /
(forall st'',
multistep_right st st' st'' ->
can_immediately_force st'').
```
Lemma enter_cif_after_trigger_porss_is_some3:
forall st other_rd_st_sig,
let '(pst, cst, _) := st in
system_invs st ->
PmtChan.status cst = PmtChan.StatusT.Triggered ->
Player.other_rd_st_sig pst = Some other_rd_st_sig ->
PmtChan.pending_rd_st_sig cst = None ->
exists st',
step SideT.Left st st' /
(forall st'',
multistep_right st' st'' ->
  can_immediately_force st'').

Listing A.4: Lemma for case 2c, when no player has presented a state yet.

Definition system_invs st :=
player_side_inv st SideT.Left /
address_eq_inv st /
address_neq_inv1 st /
address_neq_inv2 st /
(* many more omitted... *)
orrss_rd_num_gt_0_inv st /
finalized_inv st.

Listing A.5: system_invs is a conjunction of invariants that hold true when the payment channel system is first initialized and after every step.
Listing A.6: `can_immediately_force` is an auxiliary that implies `can_force` and is a little more convenient to prove.
References


