

Model Checking the Security of the Lightning Network

Matthias Grundmann, Hannes Hartenstein
KASTEL Security Research Labs
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

Abstract—Payment channel networks are an approach to improve the scalability of blockchain-based cryptocurrencies. The Lightning Network is a payment channel network built for Bitcoin that is already used in practice. Because the Lightning Network is used for transfer of financial value, its security in the presence of adversarial participants should be verified. The Lightning protocol’s complexity makes it hard to assess whether the protocol is secure. To enable computer-aided security verification of Lightning, we formalize the protocol in TLA⁺ and formally specify the security property that honest users are guaranteed to retrieve their correct balance. While model checking provides a fully automated verification of the security property, the state space of the protocol’s specification is so large that model checking becomes unfeasible. We make model checking the Lightning Network possible using two refinement steps that we verify using proofs. In a first step, we prove that the model of time used in the protocol can be abstracted using ideas from the research of timed automata. In a second step, we prove that it suffices to model check the protocol for single payment channels and the protocol for multi-hop payments separately. These refinements reduce the state space sufficiently to allow for model checking Lightning with models with payments over up to four hops and two concurrent payments. These results indicate that the current specification of Lightning is secure.

I. INTRODUCTION

Blockchain-based cryptocurrencies do not scale well with respect to their transaction throughput. One approach to improve said scalability are Payment Channel Networks – a second layer on top of a blockchain that processes payments without writing a transaction for each payment to the blockchain. A *payment channel* between two users is opened by publishing one transaction on the underlying blockchain. Once a payment channel is open, it allows for performing an unlimited number of payments between its two users without writing to the blockchain. Finally, a payment channel is closed by publishing a second transaction on the blockchain. In a payment channel *network*, the participating users are connected by payment channels and can perform multi-hop payments using a path between a payment’s sender and recipient over a set of payment channels. The first proposed payment channel network is the Lightning Network [1] built on Bitcoin [2] as the underlying layer. The Lightning Network is already used in practice and experiences a rising adoption [3].

Our goal is to verify that Lightning, the Lightning Network’s protocol, is secure, i.e., an honest user finally retrieves on the blockchain the user’s correct balance in the payment channel even if other users do not cooperate or are actively

malicious. A critical part in a payment channel protocol is the closing of a payment channel: The balance that a user finally retrieves is determined by the transactions that are written to the blockchain during the closing of the payment channel. A user can only be guaranteed to retrieve the user’s balance from the payment channel if the user is able to close the payment channel without cooperation of the other party. Lightning ensures that a user can close a payment channel independently of the other party by asserting that a user always has a valid closing transaction that could be published on the blockchain. After a payment has been performed in a payment channel, previously valid closing transactions become outdated but these transactions could still be published on the blockchain. To disincentivize a dishonest user Alice from publishing an outdated closing transaction, the other user, Bob, receives revocation secrets when the closing transaction is outdated. These revocation secrets enable Bob to punish Alice by retrieving not only his assets but also the assets of Alice if Alice would publish an outdated closing transaction.

The revocation mechanism as well as Lightning’s reliance on time and the number of involved parties make it difficult to assess whether Lightning actually fulfills the security property. Such an assessment might be facilitated by computer-aided methods. In particular, model checking can automatically verify that a protocol fulfills a property and provide a counterexample if the checked property does not hold for the protocol under investigation. Lightning is defined by an official specification [4] that describes all aspects of the protocol and partially the intuition behind the protocol. The specification is not directly usable for a security analysis because the specification contains many implementation details and is not formalized. To enable the use of model checking for Lightning, we contribute a *specification of Lightning* in the formal language TLA⁺ [5], [6] that formalizes all protocol steps and messages that users send during opening, updating, and closing a payment channel as well as for multi-hop payments. In particular, the specification models the revocation mechanism including how users exchange revocation secrets, build closing transactions, and find and react to outdated closing transactions. We chose TLA⁺ because, as a general purpose language, TLA⁺ allows for specifying arbitrary protocols and because there are tools supporting different verification methods, e.g. random state space exploration (simulation), explicit-state model checking [7], symbolic model checking

[8], and theorem proving [9]. We also contribute a *specification of the security property* of a payment channel network by defining a secure payment system in TLA⁺. The security property formally defines the intuitive property that all honest users have received their correct balances when the protocol terminates. Our security model allows parties of the protocol to become adversarial which means in our model that they omit sending messages or publishing transactions that are required to be sent by the protocol or that they publish transactions on the blockchain other than specified by the protocol. Limitations of our adversary model are that an adversarial user does not send arbitrary messages and that adversarial users do not exchange information.

However, the state space of the specification of Lightning is so large that model checking is unfeasible. We make model checking the Lightning Network possible using two refinement steps that we verify by proofs. In a first step, we prove that the model of time used in the protocol can be abstracted using ideas from the research of timed automata [10]. In a second step, we prove that it suffices to model check the protocol for single payment channels and the protocol for multi-hop payments separately. These refinements reduce the state space sufficiently to allow for model checking Lightning with the explicit-state model checker TLC [7]. We use TLC to fully explore the state space of models with payments over up to four hops and with two concurrent payments. To check the models also for larger scenarios as well as the whole stepwise refinement, we use simulation which is a lightweight alternative to model checking in which not the complete state space but only random behaviors are explored. While our approach does not give comprehensive formal correctness guarantees, it gives confidence that the current specification of Lightning is secure. We leave writing a complete formal proof and verifying all refinements using a theorem prover for future work.

We describe Lightning in more detail and give an introduction to TLA⁺ in Section II. Related work follows in Section III. Our approaches for the specification of Lightning and for the specification of the security property are presented in Section IV and Section V. In Section VI, we explain our approach for verifying that Lightning fulfills the security property and sketch the ideas behind each abstraction step. We present the results of model checking in Section VII and discuss limitations of our approach and ideas for future work in Section VIII. Section IX concludes the paper.

II. FUNDAMENTALS

In this section, we introduce Lightning and TLA⁺. We do not present all the details of Lightning that are part of our TLA⁺ formalization, but aim to give an overview of the main ideas of Lightning that contribute to the protocol’s complexity.

A. Payment Channels: Single-Hop Payments

A payment channel is a protocol between two users that enables these two users to deposit coins into the payment channel during opening, to perform payments between the

two users by updating the payment channel, and to retrieve their final funds by closing the payment channel. The protocol has to guarantee that its users can close the channel at every point in the execution of the protocol to retrieve their current balance, independently of the cooperation of the other user. Even with an actively malicious channel partner, the protocol has to ensure that an honest user cannot lose funds as long as the user actively monitors the underlying blockchain and reacts to malicious closing attempts.

A payment channel is opened in Lightning by creating a funding transaction.¹ The funding transaction has an input spending an output from the funding user (funder)² and the funding transaction has a multi-signature output that is spendable only by the two users in the channel together. Just publishing the funding transaction on the blockchain would create a dependence of the funder on the other user for spending the funding transaction’s output as the funding transaction’s output can only be spent by the two users together. To prevent such a dependence, an initial commitment transaction that spends the funding transaction’s output is created by the two users and the non-funding user sends their signature for the initial commitment transaction to the funder who only publishes the funding transaction after receiving this signature. Commitment transactions are the ‘main vehicle’ of Lightning. They are cryptographically signed by both parties and encode the state of the payment channel and can be executed on the underlying blockchain. A commitment transaction has at least two outputs: For each user, there is one output that has an amount that corresponds to the user’s current balance and is redeemable only by this user. In the initial commitment transaction, all funds are spendable by the funder.³

For a payment from one user to the other user in the channel, an HTLC (Hash Timelocked Contract) is added to the channel. An HTLC is a contract that encodes the agreement that the recipient receives a specified amount if the recipient proves knowledge of a preimage to a specified hash before a specified time has passed. The process to make a payment is as follows: The recipient of the payment draws a random value x and calculates the hash value $y = H(x)$ using a cryptographic hash function H . The recipient sends y to the sender of the payment. The sender of the payment adds an HTLC to the channel by including into a new commitment transaction a dedicated output which the receiver of the payment can spend by providing a preimage of y . The amount of coins that are part of the HTLC are deducted from the payment’s sender’s output in the new commitment transaction. After the HTLC is committed to the payment channel, the recipient of the payment fulfills the HTLC by sending the preimage x to the sender of the payment. Then, the channel is updated by

¹See BOLT 2 and BOLT 3 [4].

²Until recently, Lightning supported only single-funded channels, i.e. only one user deposits coins into the channel during opening. Our work, therefore, considers only single-funded channels (see Section VIII).

³This is a simplification; Lightning’s specification allows the funder to send a small amount to the non-funding user already in the initial commitment transaction (see [4, push_msat]).

creating a new commitment transaction without the HTLC output to remove the HTLC and, in the new commitment transaction, the HTLC's amount is added to the recipient's balance. If the recipient does not fulfill the HTLC before the timelock, the HTLC is also removed but the HTLC's amount is added back to sender's balance.

To update the channel for adding or removing an HTLC, the sender of the payment creates a new commitment transaction and sends a signature for this new commitment transaction to the payment's recipient. Now, the recipient has two valid commitment transactions, i.e., the current and the new commitment transaction, both signed by the payment's sender. As a malicious user might publish an outdated commitment transaction, outdated commitment transactions need to be revocable. As a signature to a commitment transaction cannot be undone, Lightning uses an approach that relies on incentives: A user can be punished for publishing an outdated commitment transaction. For each commitment transaction, there exists a revocation key pair. With knowledge of the private revocation key, one user can spend all outputs of the commitment transaction that the user's counterpart in the channel has published. During an update of a channel, both users send each other their signature for the new commitment transaction and reply by sending the private revocation key for the then outdated commitment transaction. As the users do not have the private revocation key for the current commitment transaction of their counterpart, they cannot punish each other for correct behavior like publishing the current commitment transaction. For the security of the protocol, it is crucial that each user has the private revocation keys for revoking outdated commitment transactions and that the other user in the channel does not have the private revocation key for the latest commitment transaction.

B. Payment Channel Networks: Multi-Hop Payments

If two users do not have a common payment channel but they are connected over a path of payment channels of other users, they can make multi-hop payments between each other. Intermediate users forward the payment over their channels and might receive a small fee for their service. To prevent intermediaries from stealing or loosing coins, it must be guaranteed for a multi-hop payment that each intermediary receives an incoming payment on one channel if and only if the intermediary forwards the payment on another channel. Also, the sender should send the payment to an intermediary if and only if the recipient receives the payment from an intermediary. Lightning uses HTLCs for multi-hop payments to achieve these security properties. As in a single-hop payment, the recipient of a payment draws a random value x , calculates the hash value $y = H(x)$, and sends y to the sender of the payment. The sender of the payment creates an HTLC with the first intermediary using y as the hash condition for the HTLC. The intermediary creates an HTLC with the next hop and each intermediary repeats this process until the last intermediary creates an HTLC with the recipient of the payment. The recipient fulfills the HTLC by

sending x to the last intermediary. By fulfilling the HTLC, the payment's recipient receives the payment's amount from the last intermediary. Again, each intermediary forwards the secret value x fulfilling the HTLCs along the route until the sender receives x and pays the first intermediary. The timelocks of the HTLCs are chosen in a descending order from the sender to the recipient, so that each intermediary has enough time to fulfill the incoming HTLC from the previous hop if the next hop fulfills the outgoing HTLC.

C. TLA⁺

We specify Lightning and the security property in TLA⁺. The Temporal Logic of Actions (TLA) [5] is a temporal logic to reason about properties of a system. The language TLA⁺ is based on TLA and used to formalize the behavior of a system. Using the explicit-state model checker TLC [7], the symbolic model checker Apalache [8], or a theorem prover (see TLAPS [9]), invariants and properties can be verified. TLA⁺ has been used repeatedly to reason about the properties of systems and protocols (see [11], [12]). We chose TLA⁺ because, as a general purpose language, TLA⁺ allows for specifying arbitrary protocols although abstractions are required for modeling cryptographic primitives (see Section IV) and because there are tools supporting different verification methods.

In TLA⁺, the state of a system is described by a set of variables v . Formally, a state is an assignment of values to variables. The state space of a system is the set of all possible states. A behavior is a sequence of states. A system is described by defining the set of valid behaviors of the system. A step $s \rightarrow t$ is a pair of successive states s and t in a behavior. An *action* is a function that takes a step and assigns a boolean value to it. If an action A assigns TRUE to a step $s \rightarrow t$, we say that the step $s \rightarrow t$ is an A step or a step of A and use the notation $s \xrightarrow{A} t$. An action is enabled in a state s if there exists an A step starting at state s .

The definition of a system in TLA⁺ is the set of all valid behaviors of the system. A system is described inductively by a set of initial states and an action that determines valid steps of the system. The set of initial states is defined as the set of states for which a formula with the name *Init* is valid. The formula *Init* has at least one conjunct for each variable of the state that describes the valid values of the variable in the initial state. The action with the name *Next* determines which steps are allowed for the system to change its state. By starting in an initial state and performing steps allowed by the action *Next*, the behaviors of the system can be generated. A system with variables v is represented as a formula $Spec = Init \wedge \Box[Next]_v$ where \Box is the *always* operator of temporal logic and $[Next]_v$ means *Next* or a stuttering step in which all variables v are unchanged. An additional conjunct may be a fairness condition $WF_v(A)$ that asserts that an A step is taken if the action A is enabled continuously. The *Next* action is typically a disjunct of multiple subactions that define different ways for the system's state to be updated. An action A is described as a conjunction of multiple conjuncts that describe the state in which the action A is enabled and the new state

that is reached by an A step. Primed variables (e.g., v') are used to describe the values of the variables in the new state and unprimed variables (e.g., v) describe the values of the variables in the current state. The subactions of $Next$ can be grouped into modules. In the TLA⁺ formalization of Lightning, we use modules to structure the specification by grouping related functionality in a module. Each module can be instantiated multiple times, parameterized with different sets of variables. We use such instantiations in the formalization to instantiate a module once for each user and channel.

The variables of a system can be internal or external. Internal variables cannot be seen by an external observer. From the outside, what a system does is described by the external variables. In TLA⁺, it can be expressed that one system implements another system. A specification S_1 implements (or, equally, refines) specification S_2 , if all external variables of specification S_1 are also external variables of specification S_2 and the way how the external variables of specification S_1 are changed in a behavior of specification S_1 also fulfills specification S_2 . Internally, the two specifications might work very differently and have different sets of internal variables. If specification S_1 implements S_2 , we use the notation $S_1 \Rightarrow S_2$.

To show that specification S_1 implements S_2 , we specify a mapping from the state space of specification S_1 to the state space of S_2 . With a slight formal incorrectness, we also say that behaviors are mapped by the mapping whereby we mean that a behavior σ_1 of specification S_1 is mapped to a behavior σ_2 of specification S_2 by applying the mapping to all states in σ_1 . The mapping is a refinement mapping if it maps every behavior σ_1 of specification S_1 to a behavior σ_2 of specification S_2 so that the values of the external variables in both behaviors are equal and σ_2 is a valid behavior of specification S_2 . If there exists a refinement mapping from specification S_1 to specification S_2 , then specification S_1 implements specification S_2 [13].

D. Explicit Real-Time Specifications

Lightning depends on time to decide whether an HTLC's timelock has passed or not. Thus, a specification of Lightning needs to be a real-time specification that models time. While there are languages especially for modeling real-time systems (e.g., KRONOS [14], UPPAAL [15]), we use TLA⁺, a general purpose language. Real-time systems can be modeled in TLA⁺ using explicit real-time specifications [16] that we define as follows. An *explicit real-time specification* has a set of variables for clocks that model the value of time. There must be at least one clock; however, there might also be multiple clocks. Because time is defined in Lightning by the height of the blockchain, we restrict all clocks to have discrete values. Progress of time is modeled by an *AdvanceTime* action (also called *Tick* in the literature) that advances each clock by the same non-negative integer value and leaves all other variables unchanged. An explicit real-time specification is similar to

the model of a timed automaton⁴ [10]. In Section VI-A, we apply research results from the area of timed automata to the Lightning specification.

III. RELATED WORK

Aspects of Bitcoin and Lightning were formally analyzed in previous work. Andrychowicz et al. [17] modeled Bitcoin contracts as timed automata and verified them using the UPPAAL model checker [15]. Setzer [18] modeled Bitcoin transactions in Agda [19]. Boyd et al. [20] created a model of a blockchain in Tamarin and analyzed Hash Timelocked Contracts, a primitive that is used by Lightning. Hüttel and Staroveški [21], [22] formalized four subprotocols of Lightning and analyzed these protocols using ProVerif for secrecy and authenticity properties. These works on Lightning are complementary to the problem definition in Section I as they show lower level properties of subprotocols but not the security of the combined protocol. Rain et al. [23], [24] formalized two subprotocols of Lightning and analyzed these protocols mechanically for game-theoretic security. Their work is complementary to the problem definition above as their formalization of the protocol assumes that an honest party actually can punish a dishonest party. This assumption is a property that we aim to prove.

The security of Lightning was analyzed before by Kiayias and Thyfronitis Litos [25]. Compared to our formalization, the protocol definition of [25] considers more details about the cryptographic aspects. They specified an ideal functionality of Lightning which is a detailed description of how the parties in a payment channel network behave. Compared to our definition of the security property, the complexity and the length of this ideal functionality make it hard to assess whether the ideal functionality corresponds to a user's intuitive expectation of security. They used the UC framework [26] to prove that Lightning securely implements this ideal functionality even in the presence of arbitrary behaving adversaries. While working on our TLA⁺ formalization of Lightning, we found two subtle flaws in the formalization of [25] of Lightning that render the formalized protocol insecure. However, we believe that these flaws can be corrected and that Lightning fulfills the ideal functionality formalized in [25]. The first flaw concerns an incomplete description of how a user reacts to maliciously published outdated transactions. The second flaw is more subtle and concerns how the data in an input is linked to the spending methods of an output that is spent by this input. A detailed description of the flaws can be found in Appendix A. While we found the first flaw by comparison of our formalization to the formalization in the paper, we found the second flaw only by model checking when we had a similar flaw in a draft of our formalization. While the specific flaws can be fixed with low effort, it is difficult and tedious to manually find such flaws in a proof. Using our approach of model checking instead, such issues can be revealed automatically.

⁴One difference is that timed automata are typically defined with real numbered values for clocks while the time in Lightning is the height of the blockchain which is always an integer.

Weintraub et al. [27] analyzed the messages exchanged during a single-hop transaction in Lightning. They inferred properties from the official Lightning specification and found by model checking an ambiguity in the official specification. This shows the need for an unambiguous formal specification of Lightning. Further, Weintraub et al. present an attack that relies on honest parties not following the Lightning protocol. Specifically, they assume that parties consider a payment processed before the payment is actually processed and that, in case of a network partition, the channel parties agree out-of-band on the outcome of a payment instead of closing the channel on the blockchain after a reasonable timeout. In this work, we assume that honest parties follow the protocol. We specify explicitly at what point a payment can securely be considered processed (see Section IV) and check that from this point on an honest party is guaranteed to receive the payment. Moreover, our work is a step towards a formal and unambiguous official specification of Lightning.

Recently, Fabiański et al. [28] formally verified a simplified variant of Lightning in Why3. They also formalized the informal security property that honest users do not lose money. In contrast to our approach of defining the security property by defining the behavior of a secure system, they use a game-based definition which is more complex. While we assume an adversary with limited capabilities, they formally verified that the formalized protocol is secure even in the presence of arbitrary behavior. Their work shows that a formal verification of even a variant of Lightning considering only single payment channels without HTLCs is a challenging effort. Their work as well as our work could be used as starting point for a formal verification of a more complete model of the Lightning protocol.

IV. FORMALIZATION OF LIGHTNING

To be able to use model checking for Lightning, we need to formalize it first, because there exists only an informal official specification. In this section, we explain important aspects of our TLA⁺ formalization of Lightning to show the assumptions and the abstraction layer of the formalization. The complete TLA⁺ formalization is available on Github⁵. A detailed presentation of the official specification and our formalization can be found in Appendix B.

The TLA⁺ formalization specifies a system with an arbitrary number of users. The behavior of a user is specified as in the official specification [4] with some simplifications that we detail below. The key task of Lightning is to ensure that each user can spend the right transaction output at the right time. Therefore, our model of Lightning focuses on the protocol logic in which users exchange data to build transactions, publish transactions, and observe transactions on the blockchain. To keep the complexity at a manageable level, we do not model fees and we abstract cryptographic primitives like signatures and hash functions.

To ensure that our TLA⁺ formalization of Lightning captures the behavior of Lightning as closely as possible, the TLA⁺ formalization follows the structure of the official specification. We make use of the same identifiers for messages as in the official specification, and the states of HTLCs in the formalization can be mapped to those used in Core Lightning⁶, an implementation of Lightning.

Lightning uses primitives such as signatures and hash functions that are not directly available in TLA⁺. For the formalization, we make the perfect cryptography assumption that the adversary cannot break cryptographic primitives, and we use a symbolic representation of cryptographic keys and signatures as done in previous work (e.g., [17], [29]). We abstract these primitives by focusing on their relevant properties that are used in Lightning. For example, Lightning is based on the assumption that a hash function is deterministic and easy to evaluate given the preimage but cannot be reverted given a hash value and that two different inputs to a hash function result in two different outputs. In the TLA⁺ formalization, the preimages that are used for multi-hop payments are not randomly generated but are deterministically assigned based on the id of the associated payment. We model the hash value of a preimage to be equal to the preimage itself and distinguish hashes and preimages by the names of the variables in which a preimage or hash value is stored. In Bitcoin, transaction identifiers are defined as a hash over the transaction. In the formalization, we model transaction identifiers by drawing a new unique value for each transaction when the transaction is created and including that identifier in the transaction.

The TLA⁺ formalization contains a model of the blockchain and all transaction types used in Lightning defining the conditions how each transaction output can be spent. We model publishing a transaction on the blockchain by a single step that happens instantly, i.e., we assume that users have blockchain connectivity and we make the simplifying assumption that each transaction to be published is included in the next block being created. For the communication between users, we model that messages are delivered reliably and in order but can be arbitrarily delayed.

In Lightning, the height of the blockchain is used as logical time that is relevant for the timeout of HTLCs. We refer to the height of the blockchain as time, which is formalized as a variable that is advanced by arbitrary integer steps. Thus, the specification is an explicit real-time specification (see Section II-D). Some actions in the protocol are urgent (see [30]) meaning that they need to happen before a certain point in time, e.g., a user has to fulfill a HTLC before the HTLC's timeout. We model this by letting each user specify deadlines and not letting time advance beyond a deadline until a step is taken that removes the deadline.

As required by the security model of Lightning, the formalization models that users can be adversarial. In principle, there are four types of adversarial behavior: (1) Omitting sending a

⁵<https://github.com/kit-dsn/lightning-tla>

⁶https://github.com/ElementsProject/lightning/blob/master/common/htlc_state.h

message to another user. (2) Omitting publishing a transaction on the blockchain. (3) Sending a message to another user with arbitrary content. (4) Publishing transactions on the blockchain other than specified by the protocol. The adversary model for the TLA⁺ formalization allows the adversary to omit sending messages (1) or publishing transactions (2). Also, the adversary is allowed to create and publish certain transactions other than specified by the protocol (4). Transactions published by an adversary are only relevant if they spend an output of a transaction that is related to the payment channel. In the formalization, an action models that the adversary publishes transactions in two ways: First, by finding all outputs that are spendable for the adversary and publishing a new transaction that spends the output to an output that is owned by the adversary. Second, by signing and publishing a transaction that the adversary has already received the other user's signature for (e.g., an outdated commitment transaction). We do not model that the adversary sends messages with arbitrary content (3) because this would significantly increase the specification's complexity. In practice, the effect of the adversary sending messages with arbitrary content is limited because users verify every message they receive. Messages that have an invalid payload or that are received at an invalid point in the protocol execution are ignored. To check the verification of messages, we explicitly model the verification of every received message and the message's payload, e.g., the verification of signatures and preimages.

We model that any user in the specification can become adversarial. However, a limitation of our adversary model is that we do not allow information exchange between adversarial users which would model a single adversarial entity controlling multiple users. This restriction simplifies the verification of the specification and we consider it future work to extend the specification with a broader adversary model.

V. SECURITY PROPERTY

Our goal is to model check the security of Lightning. Our notion of security is captured by the following informal definition. We define a user as being honest if the user behaves as required by the specification of Lightning.

Definition 1 (informal security). *An honest user will finally get paid out at least the user's correct balance.*

To make this definition precise, we formalize the security property. One approach to define security are low-level invariants and properties such as 'a user's balance is always positive' or 'if a user has received the preimage for an outgoing HTLC, the associated incoming HTLC of the user will be fulfilled'. While formalizing the specification of Lightning, we regularly used such properties to check the draft of the specification for flaws. However, it is hard to tell for such properties whether they are actually required for the protocol to be secure. Also, it is hard to tell whether these low-level properties imply what a user intuitively expects as a security property.

The informal definition implicitly concerns four variables: 1. Whether a user is honest. The security property only applies

MODULE *IdealPaymentNetwork*

VARIABLES *BlockchainBalances*, *ChannelBalances*,
Payments, *Honest*
CONSTANTS *UserIds*, *InitialPayments*, *Numbers*

IdealUser(*user*) \triangleq INSTANCE *IdealUser* WITH
UserId \leftarrow *user*,
ChannelBalance \leftarrow *ChannelBalances*[*user*],
BlockchainBalance \leftarrow *BlockchainBalances*[*user*],
Payments \leftarrow *Payments*[*user*],
Honest \leftarrow *Honest*[*user*]

IdealPayments \triangleq INSTANCE *IdealPayments*

Spec \triangleq
 $\wedge \forall \text{user} \in \text{UserIds} : \text{IdealUser}(\text{user})! \text{Spec}$
 $\wedge \text{IdealPayments}! \text{Spec}$

Fig. 1. Formal definition of the security property as an idealized payment network. The module *IdealPaymentNetwork* specifies that each user behaves as specified by the module *IdealUser* (see Fig. 2) and that the users' views on which payments have been processed are consistent as specified in the module *IdealPayments* (see Fig. 3).

to a user who follows the protocol. 2. A user's balance in a channel which defines the correct balance that a user expects to have. This balance is affected by the payments that are sent and received. 3. A user's view on the status of the user's payments, i.e., whether a payment has been sent or received. 4. A user's balance on the blockchain which are the funds that a user can directly access. The amount that a user is finally paid out is added to this balance. To formalize the informal definition, we use TLA⁺ to define how these four variables are allowed to change.

We formalize the security property of Lightning by defining the behavior of a secure payment system. The full specification of the security property is shown in Figs. 1 to 3. The security property has four variables *Honest*, *ChannelBalances*, *Payments*, *BlockchainBalances* matching the variables described above. The security property describes which changes of these variables are allowed. The definition of the security property is divided into three modules. The module *IdealUser* (see Fig. 2) defines four actions *Deposit*, *Pay*, and *Withdraw* that describe how the variables of single user can change. The module *IdealPayments* (see Fig. 3) ensures that the users' views on which payments have been processed are consistent. The module *IdealPaymentNetwork* (see Fig. 1) connects the two other modules by defining that for all users in the payment network the specification of the module *IdealUser* must hold and that the specification of the module *IdealPayments* must hold.

Initially, the variables are initialized (see Fig. 2) so that each user has a channel balance of zero, an integer balance on the blockchain, and a set of payments that the user wants to send or receive. The variable *Honest* specifies for each user whether the user is honest or dishonest.

The action *Deposit* describes a deposit by defining that a user's blockchain balance may decrease by a certain amount and the user's channel balance increases by the same amount.

MODULE *IdealUser*

EXTENDS *Integers*, *SumAmounts*
 VARIABLES *BlockchainBalance*, *ChannelBalance*,
 Payments, *Honest*
 CONSTANTS *UserIds*, *UserId*, *InitialPayments*, *Numbers*
 ASSUME $Numbers \subseteq \mathbb{N}$

Init \triangleq
 $\wedge BlockchainBalance \in Numbers$
 $\wedge ChannelBalance = 0$
 $\wedge Payments = \{pmt \in InitialPayments :$
 $\vee pmt.sender = UserId$
 $\vee pmt.receiver = UserId\}$
 $\wedge Payments \in \text{SUBSET}[amount : Numbers, id : Numbers,$
 $sender : UserIds, receiver : UserIds,$
 $state : \{\text{"NEW"}, \text{"ABORTED"}, \text{"PROCESSED"}\}]$
 $\wedge Honest \in \{\text{TRUE}, \text{FALSE}\}$

Deposit \triangleq
 $\wedge \exists amount \in 1 \dots BlockchainBalance :$
 $\wedge BlockchainBalance' = BlockchainBalance - amount$
 $\wedge ChannelBalance' = ChannelBalance + amount$
 $\wedge ChannelBalance' \in Numbers$
 $\wedge \text{UNCHANGED} \langle Payments, Honest \rangle$

Pay \triangleq
 $\wedge \exists P \in \text{SUBSET} \{pmt \in Payments : pmt.state = \text{"NEW"}\} :$
 $\exists r \in [P \rightarrow \{\text{"ABORTED"}, \text{"PROCESSED"}\}] :$
 $\wedge Payments' = (Payments \setminus P)$
 $\cup \{[p \text{ EXCEPT } !.state = r[p]] : p \in P\}$
 $\wedge \text{LET } PP \triangleq \{p \in P : r[p] = \text{"PROCESSED"}\}$
 $rBal \triangleq \text{SumAmounts}(\{p \in PP :$
 $p.receiver = UserId\})$
 $sBal \triangleq \text{SumAmounts}(\{p \in PP :$
 $p.sender = UserId\})$
 IN $\wedge ChannelBalance - sBal \geq 0$
 $\wedge ChannelBalance' = ChannelBalance$
 $+ rBal - sBal$
 $\wedge ChannelBalance' \geq 0$
 $\wedge \text{UNCHANGED} \langle Honest, BlockchainBalance \rangle$

Withdraw \triangleq
 $\wedge BlockchainBalance' \in Numbers$
 $\wedge BlockchainBalance' \geq BlockchainBalance$
 $\wedge \exists amount \in 0 \dots ChannelBalance :$
 $\wedge ChannelBalance' = ChannelBalance - amount$
 $\wedge Honest \implies BlockchainBalance' \geq$
 $BlockchainBalance + amount$
 $\wedge \text{UNCHANGED} \langle Payments, Honest \rangle$

Next $\triangleq Deposit \vee Pay \vee Withdraw$
vars $\triangleq \langle BlockchainBalance, ChannelBalance,$
 Payments, *Honest* \rangle

Spec \triangleq
 $\wedge Init$
 $\wedge \Box [Next]_{vars}$
 $\wedge \text{WF}_{vars}(ChannelBalance > 0 \wedge Honest \wedge Withdraw)$

Fig. 2. Part of the security property that defines how a user's variables are allowed to change. The action *Deposit* specifies how a user deposits funds into the channel. *Pay* specifies that payments are either processed or aborted and the channel balance is updated accordingly. *Withdraw* specifies that the channel balance is withdrawn to the blockchain. The fairness condition ensures that a user will withdraw if the user's channel balance is positive.

MODULE *IdealPayments*

EXTENDS *Integers*
 VARIABLE *Payments*
 CONSTANTS *UserIds*, *Numbers*

Init \triangleq
 $\wedge \forall user \in UserIds :$
 $Payments[user] \in \text{SUBSET}[amount : Numbers,$
 $sender : UserIds, receiver : UserIds, id : Numbers,$
 $state : \{\text{"NEW"}, \text{"ABORTED"}, \text{"PROCESSED"}\}]$

Pay \triangleq
 $\wedge \forall user \in UserIds :$
 $\vee \text{UNCHANGED } Payments[user]$
 $\vee \exists P \in \text{SUBSET} \{p \in Payments[user] :$
 $p.state = \text{"NEW"} :$
 $\wedge \exists r \in [P \rightarrow \{\text{"ABORTED"}, \text{"PROCESSED"}\}] :$
 $\wedge \forall p \in P :$
 $(r[p] = \text{"PROCESSED"} \wedge p.sender = user)$
 $\implies \exists rp \in Payments'[p.receiver] :$
 $\wedge rp.id = p.id$
 $\wedge rp.state = \text{"PROCESSED"}$
 $\wedge Payments[user]' = (Payments[user] \setminus P)$
 $\cup \{[p \text{ EXCEPT } !.state = r[p]] : p \in P\}$

Spec $\triangleq Init \wedge \Box [Pay]_{Payments}$

Fig. 3. Part of the security property that defines that the sender of a payment may only see the payment as processed if the receiver of the payment sees the payment as processed.

The action *Withdraw* describes a full withdraw by defining that a user's channel balance may be set to 0 and, for an honest user, the user's blockchain balance must increase by at least the user's channel balance. For dishonest users, it is only guaranteed that a dishonest user's blockchain balance does not decrease. The action *Pay* of the module *IdealUser* describes the execution of a set of payments by defining that the sending users' channel balances are decreased by the amounts of sent payments and the receiving user's channel balances are increased by the respective amounts. Payments can be aborted keeping channel balances unchanged. Intuitively, one expects from a secure payment network that the sender of a payment sees the payment as sent (and the payment's balance deducted from the user's channel balance) only if the receiver of the payment sees the payment as received. This condition is enforced by the action *Pay* in the module *IdealPayments* which defines that a payment can only be sent if the receiver of the payment has already received the payment or receives the payment in the same step. Further, the module *IdealUser* defines a fairness condition ensuring that the system does not terminate before all users have been paid out.

In the next section, we describe how we verify that Lightning implements the security property as defined above. In our formalization of Lightning, the *BlockchainBalances* variable is refined as the sum of unspent transaction outputs on the blockchain that a user can exclusively spend. In the view of each user, the state of a payment changes from *NEW* to *PROCESSED* when the corresponding HTLC is fulfilled. Because we verify that using Lightning with this definition of a

processed payment fulfills the security property, we also clarify how Lightning can securely be used: Honest users who sell a product in exchange for a payment can securely deliver the product once they have fulfilled the corresponding incoming HTLC and do not need to wait for the fulfilled HTLC to be removed from the channel.

Because our specification of the protocol allows for adversarial behavior, the result that the protocol specification implements the secure payment system means that no modeled adversarial behavior can break the security property, i.e., the countermeasures implemented in the protocol are sufficient.

VI. MODEL CHECKING THE SECURITY OF LIGHTNING

In this section, we give an overview of how we make it possible to model check that the TLA⁺ specification of Lightning (see Section IV) fulfills the security property (see Section V). To this end, we construct a stepwise refinement from the specification of Lightning to the security property. The structure of this stepwise refinement is graphically shown in Fig. 4.

Our goal is to model check that the TLA⁺ formalization of Lightning fulfills the security property, i.e., that specification (I) implements specification (V). The idea of the stepwise refinement is to use multiple abstractions to show that, transitively, the specification of Lightning implements the security property. The structure of our approach as well as of this section is as follows. Focusing on the first abstraction ① in Section VI-A, we prove that specification (I) implements specification (II) in which the progress of time is modeled more efficiently. In Section VI-B, we abstract from the details of the payment channel protocol by defining specification (III) in which channels are updated in idealized steps (see ②). We prove the abstraction ② by showing that each individual channel in specification (II) refines a channel in specification (III). These individual channels are modeled in specifications (IIa) and (IIIa) and we check the refinement ②a between these specifications by model checking. In Section VI-C, after another abstraction ③ to optimize the modeling of time, we use model checking to check (see ④) that specification (IV) implements the security property defined in specification (V). In this section, we present the ideas behind the proofs and briefly sketch the proofs. The full proofs can be found in Appendices C to G.

A. Improved Model of Time

The protocol as specified in specification (I) is too complex even for model checking because of the large number of possible states of the protocol. One reason for the huge state space is the modeling of time. In Lightning, time is defined by the height of the blockchain. There are many states that only differ by their value of time and are equivalent in the sense that they have the same futures. For example, consider a state s in which the value of time is 1 and there exists a single HTLC with a timelock of 10. Consider a state s' that is equal to state s except that the value of time in state s' is 2. Further, assume that the only conditions in the specification that depend

on the value of time are conditions that verify whether an HTLC's timelock has passed. Then, the same future states can be reached from states s and s' because, for a condition whether the HTLC's timelock has passed, it does not matter whether the value of time is 1 or 2. Consequently, it suffices to consider only one of the states s and s' during model checking. This equivalence of the states s and s' is captured by the following definition of bisimilarity [31] for explicit real-time specifications as defined in Section II-D.

Definition 2 (Bisimilarity). A bisimulation is a binary relation \mathcal{R} over states so that for all $(s, s') \in \mathcal{R}$ it holds that:

- 1) If $s \xrightarrow{A} t$ for some state t and an action A , then $s' \xrightarrow{A} t'$ for some state t' and $(t, t') \in \mathcal{R}$.
- 2) If $s' \xrightarrow{A} t'$ for some state t' and an action A , then $s \xrightarrow{A} t$ for some state t and $(t, t') \in \mathcal{R}$.

If $(s, s') \in \mathcal{R}$, we say that the states s and s' are bisimilar.

Intuitively, two states s and s' are bisimilar if for every step that is possible from state s there exists a matching step of the same action starting at state s' and the reached states are bisimilar. As we defined the *AdvanceTime* action of an explicit real-time specification so that it increases the clocks by any non-negative integer value, an *AdvanceTime* step from state s to state t might be matched by an *AdvanceTime* step from state s' to state t' that increases time by a different value. In the area of timed automata [10], this notion of bisimilarity defined above is usually referred to as untimed bisimilarity [32] or time-abstracting bisimilarity [33].

Prior work (e.g. [10], [33]) has proposed to improve model checking of timed automata by exploiting this type of bisimilarity. The idea behind the optimization is to group all states that are bisimilar in an equivalence class, referred to as a *zone*. A zone graph is constructed by connecting zone z_1 to zone z_2 if zone z_1 contains a state from which a step to a state of zone z_2 exists. During model checking, it suffices to explore the zone graph instead of the possibly much larger original state graph. In practice, the zone graph can be explored by considering only one representative state of each zone during model checking. E.g., we define *AdvanceTime* for exploring the zone graph to advance to the next zone by advancing the value of time to the lowest time of the states in the next zone.

In a simplified specification of the Lightning protocol without on-chain transactions, the only time-dependent conditions would check whether the timelock of an HTLC has passed. Thus, states with time values on the same side of all HTLCs' timelocks would be in the same zone. In a scenario with two HTLCs, states that are equal except for the value of time could be grouped into three zones depending on whether a state's value of time is below, between, or higher than both HTLCs' timelocks (see Fig. 5a). In Lightning however, transactions can be published with outputs carrying a relative timelock that allows the output to be spent only a certain time after the transaction has been published. Imagine a scenario with two HTLCs with the timelocks t_1 and t_2 and one unpublished transaction tx that has a relative timelock r . A state in this

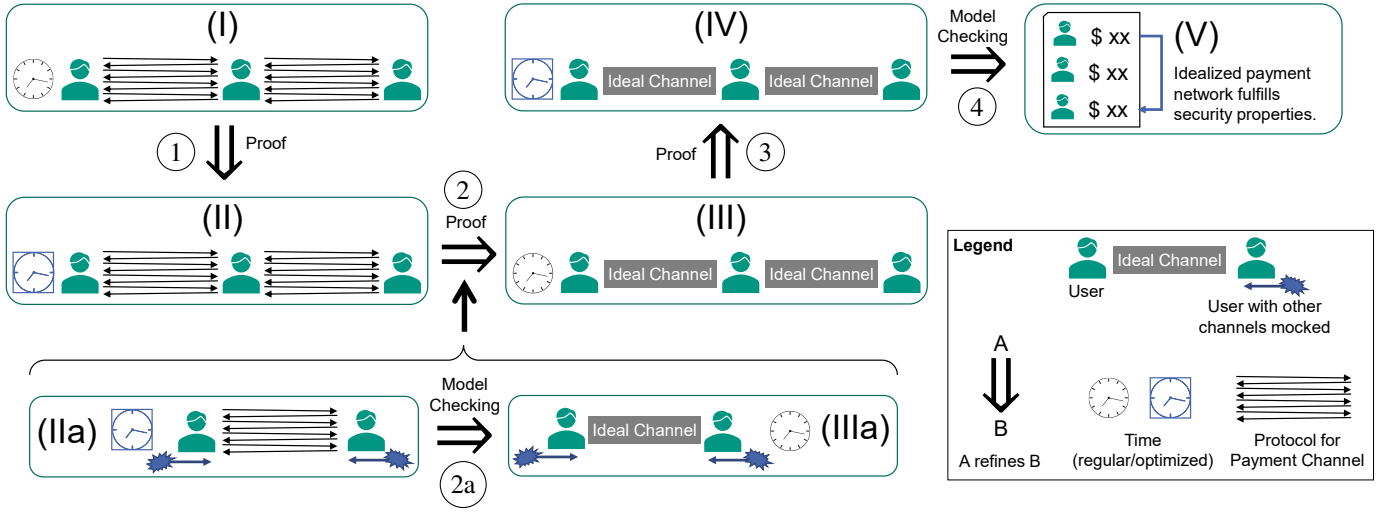


Fig. 4. Structure of the stepwise refinement to show that the Lightning protocol (*I*) refines the security property (*V*). Specification (*I*) is presented in Section IV, specification (*V*) in Section V, and the intermediate specifications in Section VI. The correctness of each step is either proven or model checked. Additionally, we check the refinement steps as well as the whole stepwise refinement using simulation, i.e., non-exhaustive model checking (see Section VII).

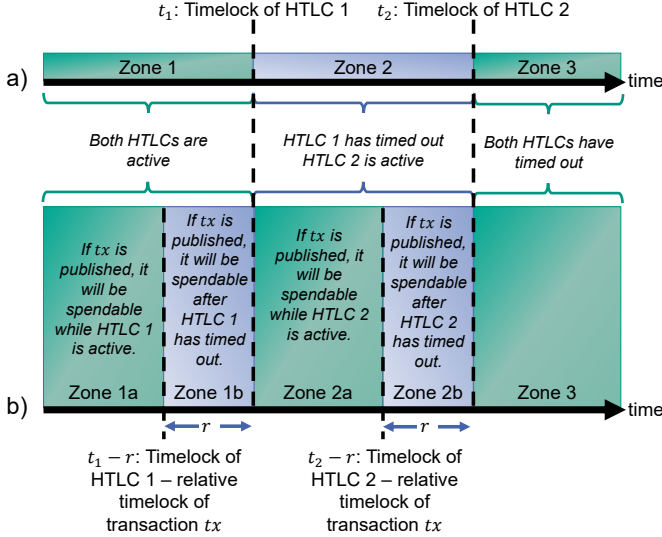


Fig. 5. Zones in which bisimilar states are grouped in a scenario a) with two HTLCs with timelocks t_1 and t_2 and b) with two HTLCs and an unpublished transaction tx with an output that has a relative timelock r .

scenario will be in one of five zones (see Fig. 5b) because zones have to be split to distinguish different futures: If the value of time in the state is below t_1 then it has to be distinguished whether the time is below $t_1 - r$ or above $t_1 - r$ because, if the transaction tx is published before $t_1 - r$, there will be a future state in which the transaction tx is spendable before time t_1 . However, if the transaction tx is published after $t_1 - r$ there will be a future state in which the HTLC with the timelock t_1 has timed out and the transaction tx cannot be spent yet. This example shows that, to distinguish in which order the relative timelocks of transaction outputs and the absolute timelocks of HTLCs are reached, a higher number of zones is required. The number of zones in the

specification of Lightning is even higher than indicated in the simplified example above because, if multiple transactions with relative timelocked outputs can be published, the order of how these transaction outputs become spendable needs to be encoded in the zone graph, too. We developed a refinement mapping that maps bisimilar states to the representatives of their respective zones. Using this mapping, the state space is reduced. However, we found this reduction still not to be sufficient to allow for efficient model checking.

A source of complexity for the time-optimized specification is that the zone graph as described above encodes information about the order of how future timelocks are reached. However, Lightning does not depend on the order of how relative timelocked outputs become spendable. Therefore, we can further simplify the zone graph used by the time-optimized specification in the following way: The subsequent zones of each zone are the zones in which the time has reached the next HTLC timelock or the age of a transaction has reached the relative timelock of one of the transaction's outputs. For better comprehensibility of the specification, we model time using multiple clocks in the specification: One variable represents the absolute value of time and, for each published transaction, a new clock is created that models the age of a transaction. When a transaction is published, a new transaction clock is started that runs with the same speed as the clock for the absolute value of time. Consequently, the *AdvanceTime* action is defined in the time-optimized specification (*II*) as follows: Time advances to the next point in time at which a new zone for the time clock starts (i.e., HTLC timelock) or the transaction age clock of a transaction (or multiple transactions) advances to the next point in time at which a new zone for this transaction age clock starts (i.e., the relative timelock of one of the transaction's outputs).

With this change in the time-optimized specification, the state space is reduced to a greater extent than with approaches

building the zone graph as the bisimulation quotient automaton (see [31], [33]). These approaches from prior work have the property that the zone graph is bisimilar to the original specification which means that for every behavior in the original specification there exists a behavior in the zone graph *and vice versa*. With our simplification of the zone graph, the time-optimized specification becomes more abstract and allows for behaviors that are not possible in the original specification. For example, a relative timelocked output of a transaction tx_2 might be spendable before a relative timelocked output of a transaction tx_1 although transaction tx_1 was published before transaction tx_2 . For the stepwise refinement, we only need that the original specification implements the time-optimized specification or, in other words, that the original specification is similar to the time-optimized specification, i.e., that for every behavior in the original specification there exists a behavior in the time-optimized specification.

We prove that the original specification (*I*) refines the time-optimized specification (*II*) by defining a refinement mapping ① from specification (*I*) to specification (*II*) and proving the correctness of this refinement mapping. The proof can be found in the appendix. In Appendix C we introduce notation and generalized approach for time optimization of explicit real-time specifications in TLA⁺. We prove the generalized time optimization in Appendix D. In Appendix E we prove that the optimization can be applied to specification (*I*). The refinement mapping ① maps a state s to a state s_R that is the zone representative of the respective zone by setting each clock in state s_R to the lowest value that the clock can have in the respective zone. In the example of Fig. 5b, a state in Zone 2a would be mapped to a state where the value of time is set to t_1 . The idea of the proof is to show that each step of specification (*I*) starting in state s is mapped to a step of specification (*II*) starting in state s_R . An *AdvanceTime* step starting in state s is either a step within the same zone or a step that advances to a new zone. A step that stays in the same zone is mapped to a stuttering step in specification (*II*). A step that advances to a new zone is mapped to a step in specification (*II*) from s_R to a next zone representative. Further, we prove for each non-*AdvanceTime* action in the specification that all steps that are possible from state s are possible from all states that are in the same zone as state s .

B. Abstraction of Protocol Steps in Payment Channels

Having applied the time optimization, the state space of specification (*II*) is still too large to be explored by model checking. Therefore, we further reduce the state space by consolidating effects of multiple protocol steps in idealized steps. We implement this approach in specification (*III*). We prove that specification (*II*) implements specification (*III*) if the protocol for a single payment channel implements the specification of an idealized channel. Thus, this abstraction step separates model checking that multi-hop payments using idealized channels refine the security property from model checking that the protocol for a single payment channel implements the specification of an idealized channel.

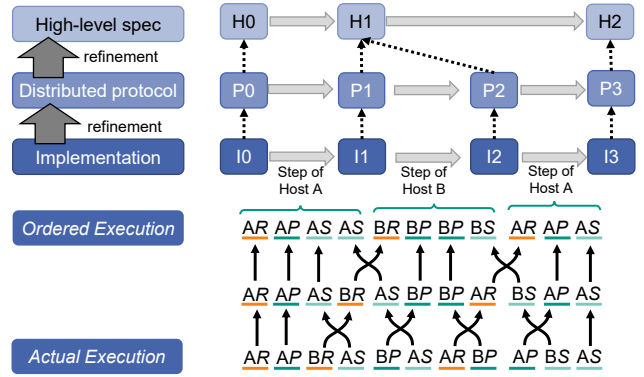


Fig. 6. Overview of the IronFleet methodology. The upper half shows the refinement steps from an implementation to a high-level specification. Each state (I0 to I3) in a behavior of the implementation is mapped to a state of the distributed protocol (P0 to P3) which is mapped to a state of the high-level specification (H0 to H2). The lower half shows how the low-level steps that each implementation step consists of are reordered. Each low-level step is abbreviated where the first letter indicates the host (A or B) and the second letter what the step does. An implementation step consists of low-level steps that receive messages (R), perform processing (P), or send a message (S). While the steps of multiple hosts can be interleaved in an actual execution, IronFleet ensures that the steps can be reordered to an ordered execution in which contiguous low-level steps match implementation steps. Figure from [34, Figures 3 and 7] (redacted).

The idealized channel specification omits some protocol details like messages that are exchanged between the parties, blockchain transactions, etc. Instead, the idealized channel specification specifies mainly how the states of HTLCs are updated as these are required for modeling multi-hop payments. For example, when opening a payment channel, the protocol specifies how messages are exchanged between the two parties; however, in the idealized channel specification, opening is modelled as a single step. Specifying the idealized channel represents a challenge: On the one hand, the specification must allow all behaviors that are possible in the protocol. This includes behaviors in which one or both users are dishonest and, for example, an HTLC is timed out although the HTLC has previously been fulfilled. On the other hand, while the specification may principally allow behaviors that are not possible in the protocol, the specification must not allow behaviors that are not possible in the protocol if these behaviors violate the security property.

A related idea has been used by Hawblitzel et al. [34], [35] for the IronFleet methodology to verify distributed systems. In the following, we present ideas used by IronFleet and explain how our approach relates to these ideas. Given an implementation for a distributed system written in Dafny [36] and a high-level specification, IronFleet is a method to automatically create a machine-checked proof that the implementation meets the high-level specification. IronFleet separates a distributed system into three layers: The high-level specification, a distributed protocol specification, and the implementation (see Fig. 6). IronFleet proves that the implementation refines the high-level specification by proving that the implementation refines the distributed protocol

specification and that the distributed protocol specification refines the high-level specification. The high-level specification specifies a centralized state machine that describes the expected externally visible behavior of a distributed system. The distributed protocol specification defines a distributed state machine that runs on each host of the distributed system. To keep the distributed protocol specification simple, the specification is abstract, e.g., it uses unbounded integers and unbounded sequences, and it is assumed that each step is an atomic step in which messages are read from the network, the state is updated and messages are sent to the network. The implementation layer is defined by imperative code that runs on each host.

To prove that the implementation of the distributed system refines the distributed protocol specification, Hawblitzel et al. first prove that the implementation for a single host refines the state machine specified by the distributed protocol specification. Because the distributed protocol specification requires atomic steps where a single atomic step consists of receiving a message, updating the host’s state, and sending a message, Hawblitzel et al. assume for this proof that each host step in the implementation is also such an atomic step. In an actual execution, such implementation steps are not guaranteed to be atomic and the low-level steps of implementation steps of different hosts might be interleaved. IronFleet uses an informal reduction argument to demonstrate that, for every actual execution, there exists an equivalent execution in which the low-level steps of each host’s implementation step are contiguous as in the atomic step. The reduction argument of Hawblitzel et al. is based on the insight that a reordering of events is valid if after reordering ‘(1) each host receives the same packets in the same order, (2) packet send ordering is preserved, (3) packets are never received before they are sent, and (4) the ordering of operations on any individual host is preserved’ [34, Section 3.6]. An example for such a reordering is shown in Fig. 6. In the following, we refer to these four rules as IronFleet reordering rules. Using the proof that the implementation for a single host refines the state machine specified by the distributed protocol specification, Hawblitzel et al. prove that a distributed system composed of multiple host implementations refines the distributed protocol of multiple hosts [34, Section 3.5].

The general approach of IronFleet is similar to our approach as our stepwise refinement from the security property (*V*) to a detailed protocol specification (*I*) corresponds to the layered proof of IronFleet from a high-level specification to an implementation. The distributed protocol layer in IronFleet corresponds to our intermediate specification (*III*) in which we abstract from protocol details. In a distributed system as considered by IronFleet, each implementation host step consists of multiple lower-level steps that are independent of other hosts and hosts communicate by exchanging messages over the network. In Lightning, payment channels are updated by multiple protocol steps that are independent of steps in other payment channels. Although it might seem counter-intuitive at first sight, we identify payment channels (and not

users) with hosts in the IronFleet methodology. A payment channel is a two party protocol that does not ‘communicate’ with other channels by exchanging messages but two payment channels affect each other by accessing the same shared variables if there is a user who participates in both payment channels. Therefore, we identify shared variables of users who participate in multiple payment channels with the network in the IronFleet methodology.

We explain how we use the reduction and refinement ideas to abstract specification (*II*) at an example: Figure 7 shows a simplified excerpt from a behavior in which a channel *B* is opened, time is advanced, and a channel *A* is updated. In the idealized channel specification (*III*), these steps are only three steps (see right side of Fig. 7). In the protocol, the abstract steps for opening and updating a channel take multiple protocol steps which might be interleaved in an actual execution (see left side of Fig. 7). We developed a refinement mapping ② that maps protocol states to states of the idealized channel specification by implicitly applying two transformations: Steps are reordered to get contiguous steps in a channel and the resulting batches of contiguous steps are replaced by idealized channel steps. We check the correctness of the refinement mapping using model checking (see Section VII).

In the following, we explain the ideas behind the construction of the refinement mapping ② that maps the protocol to idealized channels. By definition of the protocol specification, each step in the protocol is a step in one of the payment channels or a step that advances time. In the second and third column, Fig. 7 shows how each protocol step is seen by each of the two channels in the example. A protocol step of channel *A* is seen by channel *A* as the respective protocol step. Depending on whether this protocol step has an effect on other channels, the step might be seen by another channel *B* as either a stuttering step (i.e., a step that leaves all variables of the channel unchanged) or as step that changes some variables of channel *B*. We refer to the later steps as environment steps because, from the perspective of channel *B*, the variables of the channel *B* are changed not by the channel itself but by the channel’s environment. A step that advances time is an environment step in all channels. For the reordering of steps, our refinement mapping considers the IronFleet reordering rules. However, as these rules are formulated for a message passing model, they need to be adjusted for our shared variable model. The first rule ‘each host receives the same packets in the same order’ can be rephrased as ‘all environment steps are in the same order’. The third rule ‘packets are never received before they are sent’ is not necessary because each step that changes a variable used by multiple channels is simultaneously both a sending step from the perspective of the channel performing the step and a receiving step from the perspective of an inactive channel whose variables are changed by the step. The second rule ‘packet send ordering is preserved’ and the fourth rule ‘the ordering of operations on any individual host is preserved’ can be rephrased as ‘all protocol steps of a channel are in the same order’. Consequences of these reordering rules

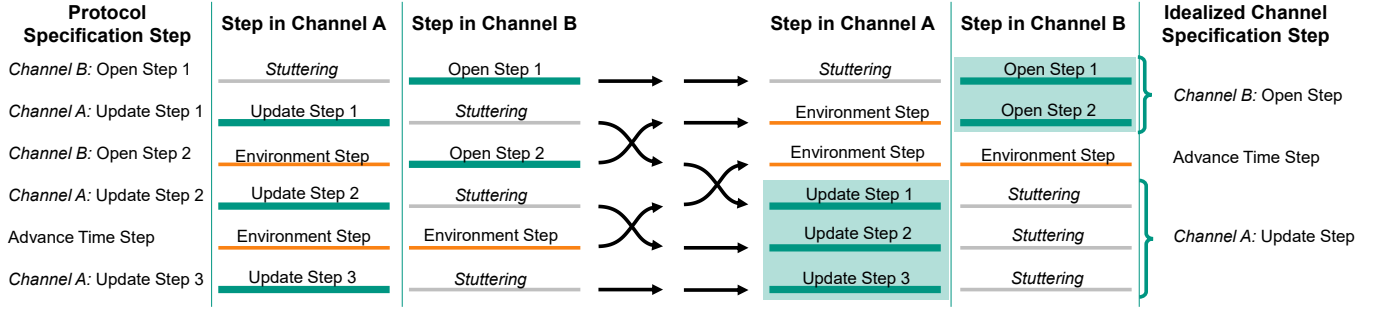


Fig. 7. In the protocol specification (II), steps for opening and updating two channels might be interleaved (left column). The refinement mapping ② from specification (II) to (III) abstracts such interleaved steps to idealized steps (right column). Thereby, the specification’s complexity is significantly reduced. The second and third columns show how each protocol step is categorized for both channels as either a step in this channel (green), an environment step (orange) that is a step of time or another channel but has an effect on the channel, or a stuttering step (gray) that has no effect on the channel. Following the reordering rules, these steps can be reordered so that in each channel multiple steps are contiguous. These contiguous steps are replaced by an abstract step of an action of the idealized channel specification (III) (right column). Note that a reordering is shown also in Fig. 6 but the representation here is rotated.

are that, from the perspective of one channel, a stuttering step can be swapped with any other step and that a protocol step and an environment step can be swapped unless the protocol step affects another channel, i.e., the step is an environment step in another channel. The reordering strategy used by our refinement mapping is that environment steps are swapped to happen before protocol steps until a protocol step is reached that ends a batch of protocol steps that is replaced by a single idealized channel step.

To verify the correctness of the refinement mapping, we use model checking and a proof (see Appendix F). Comparable to the proof of the IronFleet methodology [34], we first check for a single payment channel that the protocol specification implements the idealized channel specification with a single channel. We define specifications (IIa) and (IIIa) that describe the behavior of only a single payment channel but include a module that mocks the channel’s environment, i.e. the module specifies the effects of environment steps that other channels can have on the specified channel. Due to the environment mocking module, specifications (IIa) and (IIIa) describe all the steps the specified payment channel can take when the payment channel is part of a system with other payment channels. We prove that the environment module mocks all possible steps by other channels in Appendix F2. We specified a complex refinement mapping ②a from specification (IIa) to specification (IIIa) for the reordering and abstraction of protocol steps to idealized channel steps. We checked the refinement mapping ②a using model checking (see Section VII). We prove the refinement mapping ② in the following way: We define a refinement mapping ② from specification (II) to specification (III) that uses the refinement mapping ②a to map each individual channel from the protocol to an idealized channel. We prove that by mapping each state of specification (II) with the refinement mapping ② to a state of specification (III), each step of specification (II) is mapped to a state of specification (III).

C. Refinement of Security Property

To facilitate the refinement mapping ② from specification (II) to specification (III), specification (III) is defined as a real-time specification in which time can advance by arbitrary integer numbers instead of using the improved model of time. To allow for an efficient model checking, we apply the same optimization for time as used above by defining specification (IV) where bisimilar states are grouped in a zone (see Section VI-A). By a proof (see Appendix G) analogously to the proof of Section VI-A, specification (III) implements specification (IV) and, by transitivity, specification (I) implements specification (IV).

Finally, we can show that specification (IV) using idealized channels implements the idealized functionality defined in specification (V). We check this refinement by defining a refinement mapping ④ and checking the refinement mapping using model checking (see Section VII). The definition of the refinement mapping ④ is straightforward because all variables of specification (V) are also variables of specification (IV).

In the following section, we present our results of model checking. While the refinements introduced above significantly reduce the state space to explore, still only small finite models can be checked completely. To the extent that we could verify the refinements by model checking, we conclude from the refinement steps described above that specification (I) implements specification (V), i.e. the specification of Lightning is an implementation of an idealized functionality for a payment network and fulfills the security property.

VII. RESULTS OF MODEL CHECKING

We check the refinement mappings ②a and ④ using model checking. Additionally, we check our manual proof steps ①, ② and ③ and the whole stepwise refinement by simulation.

To model check the refinement mapping ②a from specification (IIa) to specification (IIIa), we use the explicit state model checker TLC that explores all reachable states, calculates the refinement mapping on these states and verifies that the mapped states and steps fulfill specification (IIIa). Specification (IIa) models two users and a payment channel

TABLE I
MODEL CHECKING OF REFINEMENT MAPPING (2A) FROM SPECIFICATION (IIa) TO SPECIFICATION (IIIa)

ID	Model	# States	Runtime
C1	Payment from user A to user B	$\sim 10^5$	~ 3 min
C2	Payment from user A over B to C	$\sim 10^5$	~ 11 min
C3	Payment from user C over A and B to D	$\sim 10^5$	~ 11 min
C4	Two payments: Payment from user A to B and payment from user B over A to C	$\sim 10^7$	~ 9 h
C5	Two concurrent payments from user A to B	$\sim 10^9$	~ 1 mo

and is parameterized by the information about the context of this payment channel, i.e., the other users in the payment channel network, and the payments to be processed. As there are infinitely many possible ways to parameterize specification (IIa), we only check a small selection of models that we deem representative. We model check five different models that are listed in Table I. To give an impression, the table also shows the magnitude of the number of distinct states that were explored and the time used by TLC (measured on a server with 40 physical cores). In each model, the execution starts with two users (A and B) prepared to open a payment channel and TLC explores all possible behaviors of the two users to open the channel, communicate with mocked users where applicable, process payments, and close the channel. Each checked behavior ends with the channel being closed and the two users having their funds paid out on the blockchain. The simplest model listed in Table I is a payment from user A who funded to channel to the other user. Models C2 and C3 are models in which the channel between users A and B is an intermediate hop on a payment that includes mocked users. Model C4 models two payments: A payment from user A to user B and a payment that user B sends to user C over user A as an intermediate. There are many more states to explore in model C4 as for the previous models as the two payments can partially interleave: User B has received the payment from A to B by fulfilling the corresponding HTLC. While the fulfilled HTLC is removed, User B can already send the payment to user C. Model C5 models two payment from user A to user B which is an even larger model as the two payments can interleave from the beginning. By taking about a month to model check, this model is at the limits of what we can model check in reasonable time. The main factor for this long runtime are states in which both users are dishonest. Starting from such states, many more behaviors are possible than from states in which at least one user is honest.

The models that we model check to verify the refinement mapping (4) from specification (IV) to specification (V) are listed in Table II. In all models except the last one, we model three users (A, B, and C) and two payment channels: one channel between user A and user B and the other between user B and user C. In the last model, we model four users (A to D) and three payment channels so that a payment from user A to user D is possible. In all these models, we model check

TABLE II
MODEL CHECKING OF REFINEMENT MAPPING (4) FROM SPECIFICATION (IV) TO SECURE PAYMENT SYSTEM (V)

Model	# States	Runtime
Payment from user A over B to user C	$\sim 10^5$	~ 1 min
Two payments: Payment from user A over B to C and payment from user C over B to A	$\sim 10^7$	~ 45 min
Two concurrent payments: Payment from user A over B to C and payment from user A to B	$\sim 10^7$	~ 1 h
Three payments: Payment from user A over B to C, payment from user B to A, and payment from user B to C	$\sim 10^8$	~ 13 h
Payment from user A over B and C to user D	$\sim 10^8$	~ 12 h

actual multi-hop payments and thereby verify that Lightning implements specification (V), the idealized functionality of a payment network.

Using TLC to explore the whole state space of larger models than the models we have just described becomes impractical. However, we can partially verify larger models by using TLC’s simulation mode in which the model checker starts in an initial state and chooses each next state randomly. Recent work has shown that using simulation as a ‘lightweight’ verification where more rigorous methods are not practical can be successful at finding critical flaws [37]. Using simulation, we verify the abstractions (2a) and (4) for larger models. We use also simulation to verify our structured proofs for the abstractions (1), (2), and (4) as well as to verify the whole proof that specification (I) refines the secure payment system in specification (V).

VIII. DISCUSSION

In this section, we review our approach and discuss limitations and future work.

A. Choice of Formalization Language and Tools

Protocol verifiers such as Tamarin [38] and ProVerif [39] have successfully been used for unbounded verification of a number of security protocols [40]. These verifiers could also be used to model aspects of Lightning and reason about the protocol’s properties without the limitations of finite model checking. These tools support modeling cryptographic primitives and allow for stronger adversary models. However, it is challenging to model natural numbers with addition and subtraction of variables (see [41, page 35] and [42, page 18]) which is required for modeling blockchain transactions with amounts as in our TLA⁺ specification. The generality of TLA⁺ allowed us to model all relevant aspects of Lightning. However, we had to abstract cryptographic primitives (see Section IV).

A benefit of TLA⁺ is that TLA⁺ does not restrict the way properties are proven, whether manually, using an explicit-state [7] or a symbolic model checker [8], or a tool for automated reasoning [9]. We used the explicit-state model checker TLC. The automated model checking process and the generation of counterexamples facilitated our process of

defining the intermediate specification (*III*) and the complex refinement mapping from specification (*II*) to specification (*III*). Because we used model checking, we could only verify the security for a number of four users. Theoretically, there might be attacks that only apply when there are more than four users; these attacks would not be discovered by our approach. A further consequence of the choice for model checking was that we had to restrict the adversary model to restrict the messages that an adversary can send. However, the TLA⁺ specification could also be verified using unbounded verification with a theorem prover [9]. Currently, writing such a proof seems to be too effortful. However, it will be facilitated by future advancements in assistance and automation for theorem proving.

B. Limitations and Future Work

While our work is a step towards a complete formal verification of Lightning, there are limitations not just with respect to the verification methodology as discussed above but also with respect to the model of the protocol. To formalize Lightning, we left out aspects that are not required for security. E.g., the fees that the sender of a payment pays to intermediate hops, key rotation and onion-routing for increased privacy, and route finding for multi-hop payments. Also, the model of the blockchain could be augmented by considering reorganizations and delays for transaction inclusion. Further, we assume that all parties participating in the protocol are known from the beginning and that channels are opened and then closed. We do not model that channels are reopened and that new parties join the network.

Our adversary model restricts the capabilities of an adversary by disallowing the sending of messages with arbitrary content and the exchange of information between adversarial users. While we had to make these restrictions to keep the specification's state space at a manageable level, follow up work could find optimizations that allow for making the adversary stronger.

While this work was in progress, the official Lightning specification was extended to allow for dual-funding of channels, i.e. both parties may deposit coins into a channel during opening (see [43, Channel Establishment v2]). The method we presented in this work can be used to formalize and analyze the security of this advancement as well.

The property that we model checked in this paper focuses on security. Future work could also include other properties in the idealized functionality and adapt the proof. For example, it could be shown that, assuming honest and cooperating users and timely delivery of messages, payments are guaranteed to succeed.

C. Known Attacks on Lightning

While we prove that Lightning is secure, prior work has identified several attacks on the assumptions of Lightning and properties that are not included in our security definition. Several works discuss griefing [44]–[47] and other denial-of-service attacks [48] in which no funds are stolen but the regular

operation is disturbed. In the wormhole attack [49], [50], an attacker reroutes a payment and receives the fees intended for other intermediate hops, however, the actual amount of the payment is unconcerned. Extending the TLA⁺ formalization of Lightning with fees would allow for modeling the wormhole attack. Because fees would also needed to be considered in the security property, this change would make the security property more complicated and, thus, we decided to leave modeling of fees out of scope for this paper. Further, there are attacks on privacy [51]–[58] which is a property that is not included in the security definition used in this work. Other works [59]–[62] discuss the violation of the assumption that users can timely publish a transaction on the blockchain. In practice, security flaws are also based on implementations not following the specification, e.g., by missing verification checks [63].

D. Evaluation of Protocol Modifications

Besides proving that the formalization of Lightning fulfills the security property, the TLA⁺ formalization of Lightning can also be used to test proposed modifications of Lightning. To quickly find flaws, it suffices often to model check only a subset of the specification (e.g., only a single channel) and verify just lower-level invariants and temporal properties. While such an approach cannot prove that a modification of Lightning is secure, it can accelerate protocol development by providing a short feedback loop to developers. To evaluate this idea, we introduced flaws by adapting the formalization of Lightning and verified that the introduced flaws are detected by model checking. As an example, we tested whether the, so called, second-stage transactions for HTLCs can be removed by including the conditions of the outputs of HTLC second-stage transactions directly in a commitment transaction's outputs. Verification with the model checker showed within a few minutes that this makes the protocol insecure and, thus, Lightning cannot be simplified in this way.

E. Connecting the Specification to an Implementation

There exist multiple implementations of Lightning that are actively used. While the TLA⁺ specification of Lightning is not an executable implementation, it can be used to validate the correctness of existing implementations. Cirstea et al. [64] have recently shown a lightweight way to connect implementations in imperative languages to a TLA⁺ specification. Their approach is to collect traces of program executions and to use TLC to check these traces against traces described by the corresponding TLA⁺ specification. Transferring their approach to Lightning, is an opportunity for follow up work.

IX. CONCLUSION

We have formalized Lightning and a secure payment system that captures the security property of Lightning in TLA⁺. Using stepwise refinement, we were able to model check small models which showed that Lightning implements the secure secure payment system. Therefore, the formalization can serve as a starting point for future work towards a formally verified

reference implementation of Lightning in TLA⁺. Furthermore, the approach could also be used to enable specifications of other protocols to be model checked. In particular, the abstraction of the model of time can be generalized as well as the general approach of separating model checking of local behavior and behavior in a network. Thus, our approach can be a valuable tool to analyze new versions of Lightning or similar protocols. We hope that this approach contributes to making future protocols for payment channel networks and related protocols more secure.

REFERENCES

- [1] J. Poon and T. Dryja, “The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments,” Tech. Rep., 2016.
- [2] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” Tech. Rep., 2008.
- [3] V. Barbaravičius, “Year-over-Year Data Shows Rising Lightning Network Adoption | CoinGate,” Aug. 2024. [Online]. Available: <https://coingate.com/blog/post/lightning-network-year-over-year-data>
- [4] Various. (2024) BOLT: Basis of Lightning Technology (Lightning Network In-Progress Specifications). [Online]. Available: <https://github.com/lightning/bolts>
- [5] L. Lamport, “The temporal logic of actions,” *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 872–923, May 1994.
- [6] —, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
- [7] Various. (2025) TLA+ Toolbox. [Online]. Available: <https://github.com/tlaplus/tlaplus>
- [8] —. (2025) Apalache — The Symbolic Model Checker for TLA+. [Online]. Available: <https://apalache-mc.org/>
- [9] Microsoft Research – Inria Joint Centre. (2025) TLA+ Proof System. [Online]. Available: <https://proofs.tlapl.us/doc/web/content/Home.html>
- [10] R. Alur and D. Dill, “Automata for modeling real-time systems,” in *Automata, Languages and Programming*, M. S. Paterson, Ed. Berlin, Heidelberg: Springer, 1990, pp. 322–335.
- [11] S. Merz, “Formal specification and verification,” in *Concurrency: the Works of Leslie Lamport*. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 103–129. [Online]. Available: <https://doi.org/10.1145/3335772.3335780>
- [12] R. Bögli, L. Lerena, C. Tsigkanos, and T. Kehler, “A Systematic Literature Review on a Decade of Industrial TLA+ Practice,” in *Integrated Formal Methods*, N. Kosmatov and L. Kovács, Eds. Cham: Springer Nature Switzerland, 2025, pp. 24–34.
- [13] M. Abadi and L. Lamport, “The existence of refinement mappings,” *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, May 1991.
- [14] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, “Kronos: A model-checking tool for real-time systems,” in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, A. P. Ravn and H. Rischel, Eds. Berlin, Heidelberg: Springer, 1998, pp. 298–302.
- [15] K. G. Larsen, P. Pettersson, and W. Yi, “Uppaal in a nutshell,” *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, Dec. 1997.
- [16] L. Lamport, “Real-Time Model Checking Is Really Simple,” in *Correct Hardware Design and Verification Methods*, ser. Lecture Notes in Computer Science, D. Borriore and W. Paul, Eds. Berlin, Heidelberg: Springer, 2005, pp. 162–175.
- [17] M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek, “Modeling Bitcoin Contracts by Timed Automata,” in *Formal Modeling and Analysis of Timed Systems*, A. Legay and M. Bozga, Eds. Cham: Springer International Publishing, 2014, pp. 7–22.
- [18] A. Setzer, “Modelling Bitcoin in Agda,” Apr. 2018, arXiv:1804.06398 [cs].
- [19] A. Bove, P. Dybjer, and U. Norell, “A Brief Overview of Agda – A Functional Language with Dependent Types,” in *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Berlin, Heidelberg: Springer, 2009, pp. 73–78.
- [20] C. Boyd, K. Gjøsteen, and S. Wu, “A Blockchain Model in Tamarin and Formal Analysis of Hash Time Lock Contract,” in *DROPS-IDN/v2/document/10.4230/OASlcs.FMBC.2020.5*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.
- [21] H. Hüttel and V. Staroveški, “Secrecy and Authenticity Properties of the Lightning Network Protocol,” Feb. 2020, pp. 119–130. [Online]. Available: <https://www.scitepress.org/Link.aspx?doi=10.5220/0008974801190130>
- [22] —, “Key Agreement in the Lightning Network Protocol,” in *Information Systems Security and Privacy*, ser. Communications in Computer and Information Science, S. Furnell, P. Mori, E. Weippl, and O. Camp, Eds. Cham: Springer International Publishing, 2022, pp. 139–155.
- [23] S. Rain, G. Avarikioti, L. Kovács, and M. Maffei, “Towards a Game-Theoretic Security Analysis of Off-Chain Protocols,” in *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*, Jul. 2023, pp. 107–122, iSSN: 2374-8303.
- [24] L. S. Brugger, L. Kovács, A. P. Komel, S. Rain, and M. Rawson, “CheckMate: Automated Game-Theoretic Security Reasoning,” Sep. 2023, number: 10853 Publisher: EasyChair. [Online]. Available: <https://easychair.org/publications/preprint/2G4t>
- [25] A. Kiayias and O. S. Thyronitis Litos, “A Composable Security Treatment of the Lightning Network,” in *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, Jun. 2020, pp. 334–349, iSSN: 2374-8303.
- [26] R. Canetti, “Universally composable security: a new paradigm for cryptographic protocols,” in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, Oct. 2001, pp. 136–145, iSSN: 1552-5244.
- [27] B. Weintraub, S. P. Kumble, C. Nita-Rotaru, and S. Roos, “Payout Races and Congested Channels: A Formal Analysis of Security in the Lightning Network,” in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’24. New York, NY, USA: Association for Computing Machinery, Dec. 2024, pp. 2562–2576.
- [28] G. Fabiański, R. Stefański, and O. S. T. Litos, “A Formally Verified Lightning Network,” Apr. 2025. [Online]. Available: <https://fc25.ifca.ai/preproceedings/63.pdf>
- [29] C. Boyd, K. Gjøsteen, and S. Wu, “A Blockchain Model in Tamarin and Formal Analysis of Hash Time Lock Contract,” 2020.
- [30] S. Bornot, J. Sifakis, and S. Tripakis, “Modeling Urgency in Timed Systems,” in *Compositionality: The Significant Difference*, W.-P. de Roeper, H. Langmaack, and A. Pnueli, Eds. Berlin, Heidelberg: Springer, 1998, pp. 103–129.
- [31] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT Press, Apr. 2008.
- [32] R. Alur, C. Courcoubetis, and T. A. Henzinger, “The Observational Power of Clocks,” in *CONCUR ’94: Concurrency Theory*, B. Jonsson and J. Parrow, Eds. Berlin, Heidelberg: Springer, 1994, pp. 162–177.
- [33] S. Tripakis and S. Yovine, “Analysis of Timed Systems Using Time-Abstracting Bismulations,” *Formal Methods in System Design*, vol. 18, no. 1, pp. 25–68, Jan. 2001.
- [34] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, “IronFleet: proving practical distributed systems correct,” in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP ’15. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 1–17.
- [35] —, “IronFleet: proving safety and liveness of practical distributed systems,” *Commun. ACM*, vol. 60, no. 7, pp. 83–92, Jun. 2017.
- [36] K. R. M. Leino, “Dafny: An Automatic Program Verifier for Functional Correctness,” in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds. Berlin, Heidelberg: Springer, 2010, pp. 348–370.
- [37] H. Howard, M. A. Kuppe, E. Ashton, A. Chamayou, and N. Crooks, “Smart Casual Verification of the Confidential Consortium Framework,” Oct. 2024, arXiv:2406.17455 [cs].
- [38] S. Meier, B. Schmidt, C. Cremers, and D. Basin, “The TAMARIN Prover for the Symbolic Analysis of Security Protocols,” in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds. Berlin, Heidelberg: Springer, 2013, pp. 696–701.
- [39] B. Blanchet, “Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif,” *Foundations and Trends in Privacy and Security*, vol. 1, no. 1-2, pp. 1–135, Oct. 2016.
- [40] D. Basin, C. Cremers, J. Dreier, and R. Sasse, “Tamarin: Verification of Large-Scale, Real-World, Cryptographic Protocols,” *IEEE Security & Privacy*, vol. 20, no. 3, pp. 24–32, May 2022.

- [41] T. T. Team, *Tamarin-Prover Manual*, 2024. [Online]. Available: <https://tamarin-prover.com/manual/master/tex/tamarin-manual.pdf>
- [42] B. Blanchet, “The Security Protocol Verifier ProVerif and its Horn Clause Resolution Algorithm,” *Electronic Proceedings in Theoretical Computer Science*, vol. 373, pp. 14–22, Nov. 2022, arXiv:2211.12227 [cs].
- [43] Various. (2024) BOLT 2: Peer Protocol for Channel Management. [Online]. Available: <https://github.com/lightning/bolts/blob/master/02-peer-protocol.md>
- [44] C. Pérez-Solà, A. Ranchal-Pedrosa, J. Herrera-Joancomartí, G. Navarro-Arribas, and J. Garcia-Alfaro, “LockDown: Balance Availability Attack Against Lightning Network Channels,” in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science, J. Bonneau and N. Heninger, Eds. Cham: Springer International Publishing, 2020, pp. 245–263.
- [45] A. Mizrahi and A. Zohar, “Congestion Attacks in Payment Channel Networks,” in *Financial Cryptography and Data Security*, N. Borisov and C. Diaz, Eds. Berlin, Heidelberg: Springer, 2021, pp. 170–188.
- [46] E. Rohrer, J. Malliaris, and F. Tschorsch, “Discharged Payment Channels: Quantifying the Lightning Network’s Resilience to Topology-Based Attacks,” in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, Jun. 2019, pp. 347–356, iSSN: null.
- [47] Z. Lu, R. Han, and J. Yu, “General Congestion Attack on HTLC-Based Payment Channel Networks,” in *DROPS-IDN/v2/document/10.4230/OASICS.Tokenomics.2021.2*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.
- [48] S. Tochner, A. Zohar, and S. Schmid, “Route Hijacking and DoS in Off-Chain Networks,” in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, ser. AFT ’20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 228–240.
- [49] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, “Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability,” in *Proceedings 2019 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2019.
- [50] S. Tikhomirov, P. Moreno-Sanchez, and M. Maffei, “A Quantitative Analysis of Security, Anonymity and Scalability for the Lightning Network,” in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, Sep. 2020, pp. 387–396.
- [51] J. Herrera-Joancomartí, G. Navarro-Arribas, A. Ranchal-Pedrosa, C. Pérez-Solà, and J. Garcia-Alfaro, “On the Difficulty of Hiding the Balance of Lightning Network Channels,” in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, ser. Asia CCS ’19. Auckland, New Zealand: Association for Computing Machinery, Jul. 2019, pp. 602–612.
- [52] G. van Dam, R. A. Kadir, P. N. E. Nohuddin, and H. B. Zaman, “Improvements of the Balance Discovery Attack on Lightning Network Payment Channels,” in *ICT Systems Security and Privacy Protection*, M. Hölbl, K. Rannenberg, and T. Welzer, Eds. Cham: Springer International Publishing, 2020, pp. 313–323.
- [53] E. Rohrer and F. Tschorsch, “Counting Down Thunder: Timing Attacks on Privacy in Payment Channel Networks,” in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, ser. AFT ’20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 214–227.
- [54] G. Kappos, H. Yousaf, A. Piotrowska, S. Kanjalkar, S. Delgado-Segura, A. Miller, and S. Meiklejohn, “An Empirical Analysis of Privacy in the Lightning Network,” in *Financial Cryptography and Data Security*, N. Borisov and C. Diaz, Eds. Berlin, Heidelberg: Springer, 2021, pp. 167–186.
- [55] M. Romiti, F. Victor, P. Moreno-Sanchez, P. S. Nordholt, B. Haslhofer, and M. Maffei, “Cross-Layer Deanonimization Methods in the Lightning Protocol,” in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science, N. Borisov and C. Diaz, Eds. Berlin, Heidelberg: Springer, 2021, pp. 187–204.
- [56] S. P. Kumble, D. Epema, and S. Roos, “How Lightning’s Routing Diminishes its Anonymity,” in *Proceedings of the 16th International Conference on Availability, Reliability and Security*, ser. ARES ’21. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 1–10.
- [57] A. Biryukov, G. Naumenko, and S. Tikhomirov, “Analysis and Probing of Parallel Channels in the Lightning Network,” in *Financial Cryptography and Data Security*, I. Eyal and J. Garay, Eds. Cham: Springer International Publishing, 2022, pp. 337–357.
- [58] C. Ndolo and F. Tschorsch, “On the (Not So) Surprising Impact of Multi-Path Payments on Performance And Privacy in the Lightning Network,” in *Computer Security: ESORICS 2023 International Workshops*, S. Katsikas, F. Cuppens, N. Cuppens-Boulahia, C. Lambrinoudakis, J. Garcia-Alfaro, G. Navarro-Arribas, P. Nespoli, C. Kalloniatis, J. Mylopoulos, A. Antón, and S. Gritzalis, Eds. Cham: Springer Nature Switzerland, 2024, pp. 411–427.
- [59] J. Harris and A. Zohar, “Flood & Loot: A Systemic Attack on The Lightning Network,” in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, ser. AFT ’20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 202–213.
- [60] A. Riard and G. Naumenko, “Time-Dilation Attacks on the Lightning Network,” *arXiv:2006.01418 [cs]*, Jun. 2020. [Online]. Available: <http://arxiv.org/abs/2006.01418>
- [61] T. Nadahalli, M. Khabbazi, and R. Wattenhofer, “Timelocked Bribing,” in *Financial Cryptography and Data Security*, N. Borisov and C. Diaz, Eds. Berlin, Heidelberg: Springer, 2021, pp. 53–72.
- [62] C. Sguanci and A. Sidiropoulos, “Mass Exit Attacks on the Lightning Network,” in *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, May 2023, pp. 1–3, iSSN: 2832-8906.
- [63] R. Russell. (2019) Full Disclosure: CVE-2019-12998 / CVE-2019-12999 / CVE-2019-13000. [Online]. Available: <https://lists.linuxfoundation.org/pipermail/lightning-dev/2019-September/002174.html>
- [64] H. Cirstea, M. A. Kuppe, B. Loillier, and S. Merz, “Validating Traces of Distributed Programs Against TLA+ Specifications,” in *Software Engineering and Formal Methods*, A. Madeira and A. Knapp, Eds. Cham: Springer Nature Switzerland, 2025, pp. 126–143.
- [65] A. Kiayias and O. S. Thyfronitis Litos, “A Composable Security Treatment of the Lightning Network,” 2019, report Number: 778. [Online]. Available: <https://eprint.iacr.org/2019/778>
- [66] L. Lamport, “How to write a 21st century proof,” *Journal of Fixed Point Theory and Applications*, vol. 11, no. 1, pp. 43–63, Mar. 2012.

APPENDIX

A. On the Formalization of [25]

While working on the formalization of Lightning in TLA⁺, we found the following two flaws in the formalization of [25]. While these flaws render the formalized protocol insecure, they are easy to fix and it seems that the security proof could work for the corrected protocol. The following references to figures and page numbers refer to the paper’s version on ePrint [65, version 20220217:205237].

The first flaw concerns the punishment of the publication of an outdated commitment transaction for which the protocol is specified in Fig. 37, lines 21-25 (page 64). A problem arises, for example, in the following situation: Before the current time, user Alice has sent an outgoing HTLC to user Bob. The HTLC was committed and has been fulfilled. Now, the HTLC’s absolute timelock has passed. Now, Alice has an outdated commitment transaction that commits the HTLC and Alice has Bob’s signature on the HTLC timeout transaction corresponding to that HTLC. Alice is malicious and publishes this outdated commitment transaction together with the HTLC timeout transaction which is valid because the HTLC’s absolute timelock has passed. Bob runs the protocol specified in Fig. 37 and arrives at line 22. In line 22, a revocation transaction is created whose inputs spend all outputs of the outdated commitment transaction. In the situation described, such a revocation transaction cannot be valid because the HTLC output in the outdated commitment transaction is already spent. Instead of an input referencing the outdated commitment transaction’s HTLC output, the revocation transaction must have an input that references the

output of the HTLC timeout transaction. While the protocol as formalized in Fig. 37 is incorrect, the security proof on page 90 does not mention the case that a second-stage (timeout or success) HTLC transaction might have been published for an outdated commitment transaction and, thus, the protocol seems to be correct. While the protocol can be corrected by adding a specification of how such cases are handled, it is difficult to detect such flaws by inspecting the proof manually.

For a scenario that shows the impact of the second flaw, assume that in the payment channel between users Alice and Bob there is currently an unfulfilled HTLC for a payment from Alice to Bob. The HTLC's absolute timelock passes and the HTLC times out. Bob unilaterally closes the payment channel by publishing the latest commitment transaction. The commitment transaction contains an output for the HTLC with the spending method $pt_{rev,n+1} \vee (pt_{htlc,n+1}, CltvExpiry \text{ absolute}) \vee (pt_{htlc,n+1} \wedge ph_{htlc,n+1}, \text{on preimage of } h)$ (see Fig. 40, line 8) where pt are public keys for which Alice has the private key, ph are public keys for which Bob has the private key, and $CltvExpiry$ is the HTLC's absolute timelock. Now, Alice could spend the output of the commitment transaction corresponding to the HTLC by creating a transaction with an input that uses the disjunct $(pt_{htlc,n+1}, CltvExpiry \text{ absolute})$ because the absolute timelock has passed. Bob holds the HTLC success transaction that was signed by Alice with the private key corresponding to $pt_{htlc,n+1}$ (Fig. 43, line 13). If Bob has the preimage for the HTLC, Bob can add the preimage to the HTLC success transaction and can spend the HTLC's output in the commitment transaction using the disjunct $(pt_{htlc,n+1} \wedge ph_{htlc,n+1}, \text{on preimage of } h)$ of the spending method. However, the HTLC success transaction is also valid without the preimage as it fulfills the conditions of the disjunct $(pt_{htlc,n+1}, CltvExpiry \text{ absolute})$ because the HTLC's absolute timelock has passed. Because Bob published his latest commitment transaction, Alice cannot revoke the transaction and this would result in Bob receiving the amount of the HTLC without releasing (or even without having) the preimage. One way to correct this problem is to use the possibility that the transaction model of the paper [65, Section 12] allows an output to specify a list of spending conditions and an input spending this output to reference a specific spending condition. The correction would be to transform the disjunction in Fig. 40, line 8 into a list of spending methods and add the corresponding indices to the inputs in Fig. 43, line 13. Another way is taken by the Lightning Network's specification which uses in the output's spending method for a timeout the operator `CHECKLOCKTIMEVERIFY` that verifies that a spending transaction has a certain timelock set (`locktime`). As Bob's HTLC success transaction has the `locktime` set to 0, the success transaction cannot fulfill this spending method. We found this flaw by model checking when we had a similar flaw in a draft of our formalization. We fixed the flaw in our formalization by modeling the `locktime` field for transactions and adding a validity condition modeling the operator `CHECKLOCKTIMEVERIFY`.

B. Lightning and Formalization

In this section, we present Lightning according to the official specification [4] and describe what aspects we consider for the formalization and which aspects we leave out of scope. We also explain how we model the protocol in TLA^+ . We start by explaining how Lightning implements multi-hop payments in Appendix B1. Later, we explain how payment channels are opened, updated, and closed. We give an overview of the TLA^+ formalization of Lightning in Appendix B2. We present the basic ideas behind the protocol's formalization in TLA^+ at the example of multi-hop payments in Appendix B3. This includes an explanation how sending and receiving messages and the state of an HTLC is modeled. In this section, we also discuss how random values, hash functions, transactions, and signatures are modeled. Because TLA^+ does not offer a native way to model cryptographic primitives such as hash functions and signatures, we model those primitives in an abstract way. In the real world, Lightning relies on security properties of these primitives, e.g., that it is practically impossible to find a preimage to a given hash value or that it is practically impossible to create a valid signature for a public key without knowing the associated private key. In the formalization, we assume that these security properties of the primitives hold, e.g. that an adversary is not capable of reverting a cryptographic hash function. Therefore, we can model the primitives via abstractions.

1) *Multi-Hop Payments*: This section explains how Lightning implements multi-hop payments. If a user wants to receive a payment, the user creates an invoice and sends it to the user who wants to send the payment. The invoice (see Table III) contains a payment hash for which the receiver knows the preimage and a payment secret. Further, the invoice contains additional data to describe the purpose of the payment for example. This additional data is not included in the TLA^+ formalization because it is not required for the security of the protocol. The model of an invoice used in the TLA^+ formalization contains just the hash and the payment secret.

After the payment's sender has received the invoice, the sender extracts the payment hash and the payment secret. The payment's sender chooses a route through the Lightning Network, i.e. a list of users that are connected through payment channels and whose last hop is the payment's receiver. For this paper, we leave route finding out of scope and assume that the sender receives the route as external input.

The payment's sender sends an 'update_add_htlc' message (see Table IV) to the first hop on the path. The 'update_add_htlc' message contains an id for the HTLC and the HTLC's amount, hash, and timelock. Further, the message contains an 'onion_routing_packet' that contains the information who the next hop is and an encrypted package for the next hop with the information what the hop after the next hop is and so on. The next hop receives the 'update_add_htlc' message and verifies that the amount and timelock are in expected ranges and that the onion_routing_packet can be decrypted. If a verification fails, the channel is closed. The principle that the channel is

closed when a verification fails, is used throughout Lightning. If the hop receiving the ‘update_add_htlc’ message is not the payment’s receiver but an intermediate hop: When the incoming HTLC for the payment is ‘irrevocably committed’, the intermediate hop sends an ‘update_add_htlc’ message to the next hop on the route. An HTLC is irrevocably committed for a user A if user A has received the other user’s signature on the commitment transaction containing the HTLC and has received the secrets required for revocation for all transactions that do not contain the HTLC and were signed by user A. The forwarding of the ‘update_add_htlc’ message is repeated until the payment’s receiver receives the ‘update_add_htlc’ message. If the hop receiving the ‘update_add_htlc’ message is the payment’s receiver, the receiver sends the payment’s preimage in an ‘update_fulfill_htlc’ message (see Table V) to the previous hop. In the TLA⁺ formalization, the ‘update_fulfill_htlc’ message contains only the preimage because the associated HTLC can be found by hashing the preimage and looking up the HTLC by its hash value. If the hop receiving the ‘update_fulfill_htlc’ message is not the payment’s sender, the hop forwards the payment’s preimage in an ‘update_fulfill_htlc’ message to the previous hop. The forwarding of the ‘update_fulfill_htlc’ message is repeated until the payment’s sender receives the ‘update_fulfill_htlc’ message. If the hop receiving the ‘update_fulfill_htlc’ message is the payment’s sender, the payment has been successfully performed.

If an incoming HTLC that is committed times out, a user sends an ‘update_fail_htlc’ message (see Table VI) to the previous hop. While the message contains fields for the reason of failure in Lightning, the TLA⁺ formalization contains only the id of the failed HTLC because learning the reason for failure is relevant for debugging but not for the protocol’s functionality or security. If the hop receiving the ‘update_fail_htlc’ message is not the payment’s sender: When the removal of the hop’s outgoing HTLC is irrecoverably committed or the appropriate HTLC timeout transaction is confirmed on-chain, the hop sends an ‘update_fail_htlc’ message to the previous hop. The forwarding of the ‘update_fail_htlc’ message is repeated until the payment’s sender receives the ‘update_fail_htlc’ message. If the hop receiving the ‘update_fail_htlc’ message is the payment’s sender: When the removal of the hop’s outgoing HTLC is irrecoverably committed or the appropriate HTLC timeout transaction is confirmed on-chain, the payment is finally cancelled.

2) *Formalization Overview:* We formalize Lightning in TLA⁺. The formalization describes all possible actions how a user of the payment channel initiates transactions or reacts to messages or events. In its structure, the formalization of the protocol specification follows the informal specification of the Lightning Network [4]. The formalization abstracts, however, multiple implementation details and parts that are not part of the main functionality such as fees and error messages.

We formalize Lightning in an event-based specification. In this manner, the protocol can be implemented and this approach allows for validating that the protocol is secure for every possible order of events.

TABLE III
FIELDS OF A LIGHTNING ‘INVOICE’.

Variable	Description	Formalization
timestamp	Unix Timestamp	Not required.
p	Payment hash	‘hash’ field of invoice
s	Payment secret	‘paymentSecret’ field of invoice
d	Description of purpose of payment	Not required.
m	Additional metadata	Not required.
n	Public key of the payment’s receiver.	Not required.
h	Description of purpose of payment	Not required.
x	Expiry time	Not required.
c	Minimal delta between HTLC timelocks for last HTLC	Not required because formalized as a constant default value.
f	Fallback on-chain Bitcoin address	Not required.
r	Routing information	Not required.
g	Feature bits	Not required.
signature	Signature of above fields	Not included. We instead assume that the receiver can verify the integrity of the invoice.

TABLE IV
FIELDS OF ‘UPDATE_ADD_HTLC’ MESSAGE.

Variable	Description	Formalization
channel_id	ID of the channel derived from funding transaction	Not required.
id	ID of the HTLC	‘id’ field of HTLC
amount_msat	Amount of HTLC in millisatoshi	‘amount’ field of HTLC
payment_hash	Hash of HTLC	‘hash’ field of HTLC
cltv_expiry	Timelock of HTLC	‘absTimelock’ field of HTLC
onion_routing_packet	Data for forwarding to next hop including encrypted payload for next hop.	‘dataForNextHop’ field of HTLC

TABLE V
FIELDS OF ‘UPDATE_FULFILL_HTLC’ MESSAGE.

Variable	Description	Formalization
channel_id	ID of the channel derived from funding transaction	Not required.
id	ID of the HTLC	Not required.
payment_preimage	Preimage for HTLC	‘preimage’ field

TABLE VI
FIELDS OF ‘UPDATE_FAIL_HTLC’ MESSAGE.

Variable	Description	Formalization
channel_id	ID of the channel derived from funding transaction	Not required.
id	ID of the HTLC	‘id’ field of HTLC
len	Length of the reason field	Not required.
reason	Reason why HTLC failed	Not required because the only modeled reason is timeout.

The TLA⁺ specification of the protocol consists of three modules: Two modules concern the specification of actions that a user performs for the execution of the payment channel protocol: The module HTLCUser specifies the actions concerning HTLCs for multi-hop payments, e.g., sending an invoice, creating an HTLC, fulfilling an HTLC. The module PaymentChannelUser specifies how the payment channel is created, how the payment channel is updated when a new HTLC is added or a fulfilled HTLC is persisted, how the payment channel is closed, how an adversarial user can cheat by publishing transactions on the blockchain, and how an honest user punishes a cheating user. For example, the module PaymentChannelUser includes actions for creating and sending a signature of a new commitment transaction to the other user, processing messages from the other user, or publishing a commitment transaction on the blockchain to close the channel. The third module is LedgerTime, the clock that increases the current time. Time is measured in Lightning by the block count of the Bitcoin blockchain. Thus, it is represented as an integer number and increased in integer steps. The specification puts these three modules together by instantiating the LedgerTime module, the HTLCUser module, and the PaymentChannelUser module. While the action of the LedgerTime module is a global action to advance time, the actions of the HTLCUser module and the PaymentChannelUser modules are parameterized by a user and, if applicable, a channel and the other user in the channel. Formally, the TLA⁺ specification is defined by a set of initial states and a *Next* action that describes possible steps that can lead from one state to a new state.

The TLA⁺ formalization models a system that is comprised of the users of a payment channel network and the payment channels between them. A state of the modeled system is defined by the variables that are shown in Table VII. Some of these variables describe aspects of the system in general and some model variables of a specific user of which some are for a specific payment channel of that user. The meaning and use of these variables are explained in the following sections.

The TLA⁺ formalization expects as external input the three constants NameForUserID, a sequence of modeled users, *uInitialPayments*, a set of payments to be sent by user *u*, and *uInitialBalance*, the initial balance of user *u*. Each record in *uInitialPayments* describes a payment by an id, an amount, a point in time until the payment should be processed, and a path

from the sender to the recipient. In the initial state, the variable *uNewPayments* is initialized to the value of *uInitialPayments* for a user *u* and to each payment record fields are added for storing values later, e.g., for the hash associated with the payment or a boolean value whether an invoice for this payment has been requested.

3) *Formalization of Multi-Hop Payments*: The actions of a user as described in Appendix B1 are specified as actions of the HTLCUser module. The actions in the HTLCUser module are the actions of one specific user *u* for a specific channel *c* and are parameterized by the id of channel *c*, the id of user *u*, and the id of the other user in channel *c*.

The action ‘RequestInvoice’ of the module HTLCUser is enabled if there is a payment for the user *u* in *uNewPayments* and no invoice has been requested for this payment. The action chooses such a payment, sends a message of type ‘RequestInvoice’ to the payment’s recipient and updates the payment to store that an invoice has been requested for this payment. *Sending of a message* is modeled by appending a record to a global variable ‘Messages’. The ‘Messages’ variable contains a list of all messages that are in transit. The appended message has a field that describes the message’s type, a field that states the message’s sender, and a field that states the message’s recipient and additional fields for payloads. The payload for a ‘RequestInvoice’ message is the payment id for which an invoice is requested.

The action ‘GenerateAndSendPaymentHash’ of the module HTLCUser models the reception of a ‘RequestInvoice’ message and the reply of sending an invoice to the sender of the ‘RequestInvoice’ message. The *reception of a message* is modeled by a condition that inspects the records in ‘Messages’ that have a recipient that equals the user *u*. If the first message that is sent to the user *u* has the type ‘RequestInvoice’, then the action ‘GenerateAndSendPaymentHash’ is enabled. The action has to draw random values for the preimage and the payment secret. TLA⁺ does not offer a native way to model randomly drawn values. In Lightning, the values for the preimage and the payment secret should be unpredictable and hard to guess for an adversary. In the TLA⁺ formalization, it suffices to model the adversary in a way that the adversary cannot guess the preimage and the payment secret. Random values drawn in Lightning for the preimage and the payment secret are most likely unique, i.e. drawing these values a second time randomly will most likely lead to different values. This uniqueness property of drawing random values is modeled in the following way in the TLA⁺ formalization: A random preimage and payment secret are deterministically derived from the payment’s id for which the preimage is used by adding constants to the payment’s id. Because the ids of payments are unique, this approach ensures that the preimages and payment secrets are unique as well. In Lightning, the receiver of a payment sends as part of the invoice the hash of the preimage to the payment’s sender. However, in TLA⁺ there is no native way to calculate the hash value of a variable. The reason why Lightning uses a hash function is that it should be simple to check whether a preimage and a hash correspond

TABLE VII
VARIABLES THAT ARE PART OF THE STATE OF THE TLA⁺
FORMALIZATION.

Variable	Description
Global variables	
LedgerTime	Models the current time as an integer value representing the current height of the blockchain.
Messages	Set of messages of which each user can process messages that are sent to the user.
LedgerTx	Models the blockchain as the set of all published transactions.
TxAge	Models the age of published transactions, i.e. how many blocks have been created since the transaction was published.
Variables for user u :	
u Payments	Set of payments of user u for which user u is either sender or recipient. Each payment has a status that indicates whether the payment is new, processed, or aborted.
u ExtBalance	External balance of user u . External refers to the balance on the blockchain, i.e. outside the payment channel network.
u ChannelBalance	Balance of user u inside the payment channel network. This equals the sum of the user's balance in all payment channels.
u Honest	Specifies whether the user u is honest. The value is initially set to either true or false and does not change.
u NewPayments	Set of records for payments that the user u wants to send. The module HTLC-User processes these payments by creating HTLC records that are added to variable Vars.
u PreimageInventory	Set of the preimages that the user u knows.
u LatePreimages	Preimages that have been received after the associated HTLC has timed out.
u PaymentSecretForPreimage	Function that maps for each incoming payment for which an invoice has been sent the preimage to the payment secret.
Variables for channel c :	
c Messages	Sequence of messages sent between two users of a channel. Each user can process the first message that is sent to the user.
c UsedTransactionIds	Helper variable that stores all ids that have already been used for creating transactions. This set helps to ensure that each new transaction id is unique.
Variables for user u of each channel c :	
c, u Balance	Integer that indicates the current balance of the user u in channel c .
c, u State	Protocol state the user u is in in the channel c . This implies what messages the user u expects to receive next.
c, u Vars	Record with variables of the user u in the channel c . Most importantly Vars contains the fields 'IncomingHTLCs' and 'OutgoingHTLCs' that store sets that contain a record for each incoming resp. outgoing HTLC.
c, u DetailVars	Contains variables that are only used inside the module PaymentChannelUser.
c, u Inventory	Contains keys and a set of transactions that the user u can sign and, if the transaction is valid after signing, publish on the blockchain.

to each other but it should be impossible to calculate the preimage given its hash value. As we can restrict the adversary in the specification to not calculate preimages from hashes, we use a simple approach to model a *hash function*: We use the Identity function as hash function, i.e. the hash value of a preimage equals the preimage. With this approach, it is trivial to check whether a hash corresponds to a preimage. The distinction whether a given value is a preimage or a hash, depends on the variable that a value is stored in. The formalization is carefully written so that a value of a hash is never written to a variable that contains a preimage. However, writing a hash value given the knowledge of a preimage is possible as this models the execution of the hash function. The action 'GenerateAndSendPaymentHash' sends a message modeling the invoice that contains the hash value and the payment secret to the sender. Further, the preimage and the payment secret are stored in the variables u PreimageInventory and u PaymentSecretForPreimage respectively.

The request for an invoice can also be ignored without further changes. This is modeled by the action 'IgnoreInvoiceRequest'.

The reception of the invoice by the payment's sender is modeled by the action 'ReceivePaymentHash'. The payment's sender stores the hash value and the payment secret in the record that describes the respective payment and calculates the onion package that is sent with the payment along the route. The onion package contains for each hop a value that determines the next hop, the absolute timelock for the HTLC to use with the next hop and an onion package for the next hop. For the last hop, the onion package contains the payment secret and the payment's amount.

If the payment channel c is open and ready to operate, the payment channel c 's state stored in the variable c, u State of user u is 'rev-keys-exchanged'. The action 'AddAndSend-OutgoingHTLC' sends an 'update_add_htlc' message if the channel is ready and there is a payment that fulfills the following conditions: The action is enabled for user u if the user u is the payment's sender or if the payment's incoming HTLC has been irrevocably committed (see Appendix B1). The hash for the payment must be known and the payment's timelock must be in the future. The next hop for this payment must be the other user of the payment channel c . An HTLC with the same hash must not already exist. The current balance of user u in the channel c must be at least the payment's amount. The effects of the action are that the payment is removed from the u NewPayments variable, the HTLC is added to the c, u Vars variable, and the 'update_add_htlc' message is sent. This message is sent with relation to channel c . We use the c Messages variable to model the messages exchanged between two users of a channel c . Thus, the 'update_add_htlc' message is stored in c Messages.

The reception of the 'update_add_htlc' message is modeled by the action 'ReceiveUpdateAddHTLC'. This action creates a new payment record in the user's u NewPayments variable if the payment is to be forwarded and, otherwise, verifies that the payment secret is correct and that the HTLC has the correct

amount.

If an incoming HTLC is irrevocably committed and the user u is the payment's receiver, the user u can fulfill the incoming HTLC. Whether an HTLC is irrevocably committed, is encoded in the state of the HTLC: The *state of an HTLC* is initially set to NEW when the HTLC is created in the 'AddAndSendOutgoingHTLC' action or the 'ReceiveUpdateAddHTLC' action (see Fig. 8). Once the HTLC is irrevocably committed, an action of the PaymentChannelUser module sets the HTLC's state to COMMITTED. If an HTLC is fulfilled, the HTLC's state is advanced to FULFILLED. Once the HTLC is irrevocably removed, an action of the PaymentChannelUser module sets the HTLC's state to PERSISTED. If an HTLC is failed, the HTLC's state is updated to OFF-TIMEDOUT. Once the HTLC is irrevocably removed, an action of the PaymentChannelUser module sets the HTLC's state to TIMEDOUT. To fulfill the HTLC, a user sends an 'update_fulfill_htlc' message to the other user in the payment channel. The sending of the 'update_fulfill_htlc' message is modeled by the 'SendHTLCPreimage' action. The formalization models that an HTLC can be fulfilled even after the HTLC's timeout during a grace period of a fixed length. The length of the grace period, i.e., the number of blocks to wait after an HTLC's timeout, is defined by the constant G which we set to 3 by default. Adding a grace period is suggested by the official Lightning specification but not required. We included the grace period in the formalization because the grace period creates an interesting situation that is relevant for the protocol's security because during the grace period the HTLC can be timed out as well as fulfilled. If an HTLC is fulfilled after its timeout, the hash of the HTLC is stored in the field 'FulfilledAfterTimeoutHTLCs' of the variable $c, uVars$.

The action 'ReceiveHTLCPreimage' models the reception of an 'update_fulfill_htlc' message. If the preimage is received after the HTLC's timeout and the grace period have passed, the payment might still be successful but it can also be aborted because the preimage reached the user too late. The fact that the HTLC's preimage was received late is stored by adding the preimage to the set stored in the variable $uLatePreimages$.

Failing an HTLC is modeled by the action 'SendHTLCFail'. This action sends a 'update_fail_htlc' message and updates the failed HTLC's state to OFF-TIMEDOUT. The action 'ReceiveHTLCFail' receives the 'update_fail_htlc' message and updates the state of the receiver accordingly.

4) *Keys and Funding-, Commitment- and HTLC-Transactions*: In this section, we present the keys that are used in Lightning and what they are used for. To reduce the risk of being tracked by third parties, each user in Lightning has a set of private and public keys of which each key is used for one specific purpose. Further, keys are rotated with every commitment transaction to prevent leaking information to a third party that gets to know multiple commitment transactions. As our analysis focuses on the security property that users finally receive their correct balance and not on privacy leaks, we model these keys by a single key pair per user.

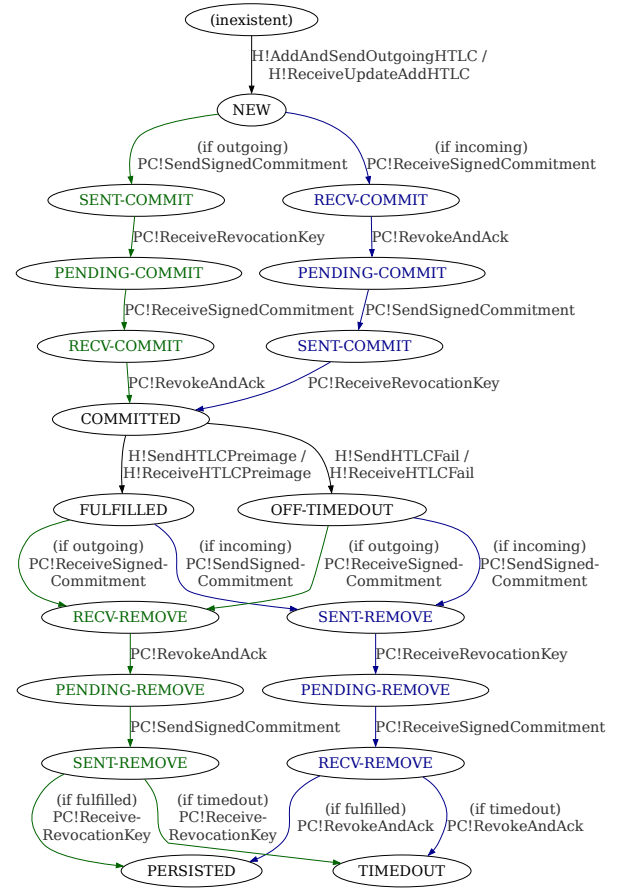


Fig. 8. Flow chart of HTLC states. The actions with the prefix H! are actions of the HTLCUser module (see Appendices B1 and B3); those prefixed with PC! are actions of the PaymentChannelUser module (see Appendices B9, B11 and B13). Outgoing HTLCs follow the path printed in green; incoming HTLCs follow the path printed in blue.

5) *Keys and Key Derivation*: Lightning makes use of multiple keys to build transactions. Here, we explain these keys as they are used in Lightning. For the formalization, we use a simplified model that we will present below. Each user has the following set of keys according to the Lightning specification. From one user's perspective, the user's own keys are prefixed by 'local' and the other user's keys are prefixed by 'remote'.

- **funding_pubkey**: A user's public key that is used to lock the output of the funding transaction. The appropriate private keys of both users are required to create a signed commitment transaction.
- **localpubkey (remote_pubkey)**: Public key used to lock an output that is spendable by the user (other user) in the commitment transactions created by the other user (user).
- **local_delayedpubkey (remote_delayedpubkey)**: Public key used to lock an output that is spendable by the user (other user) in the commitment transactions created by the user (other user).
- **local_htlcpubkey (remote_htlcpubkey)**: Public key used

to lock an output that is spendable by the user (other user) in HTLC transactions.

- **revocationpubkey**: Public key used to lock an output that is spendable when the transaction should be revoked.

Most keys are rotated with every commitment transaction so that third parties (e.g., watchtowers) cannot link multiple commitment transactions to the same channel. To simplify the key management, the different public keys are derived from a basepoint per type of key and a point per transaction (see Fig. 9a). For each of the keys prefixed by local or remote, there is a basepoint for which the secret is kept by the local party. The keys (local/remote)pubkey, (local/remote)_delayedpubkey, (local/remote)_htlcpubkey are derived from the from the per_commitment_point and from the (local/remote)_payment_basepoint, (local/remote)_htlc_basepoint, (local/remote)_delayed_payment_basepoint respectively. The private keys are derived from appropriate basepoint secrets and the per_commitment_point (see Fig. 9b). The revocationpubkey is derived from the local user’s revocation_basepoint and from the remote user’s per_commitment_point (see Fig. 9c). The associated private key, called revocationprivkey, can be derived from the local user’s revocation_basepoint_secret and the remote user’s per_commitment_secret. The idea behind this is that both users can derive the revocationpubkey but initially no user can derive the revocationprivkey. However, the per_commitment_secret can be shared so that the local user can derive the revocationprivkey to spend outputs of published outdated commitment transactions. However, as all other keys that are derived using the per_commitment_secret depend on an additional basepoint secret, the local user cannot derive other private keys of the remote user. The funding_pubkey is a regular key pair.

6) *Formalization of Keys*: To keep the specification simple, we model the funding_pubkey and the various keys that are derived from a basepoint of a user and the user’s per_commitment_point as the same key pair. What is left are the revocation public keys that are derived from the revocation_basepoint of one user and the *other* user’s per_commitment_point. These key pairs need to be rotated with every commitment transaction because the users exchange the secrets required to derive the private keys. For modeling *asymmetric key pairs*, we use a similar approach as for modeling preimages and hashes: We use the same value for the private and the public key and distinguish between them by the name of the variable that assumes a value. This user key which models all user specific key pairs used in Lightning is modeled as a symbolic value per user. We model revocation keys by a record that contains a symbolic value that specifies the creator of the key and a numerical index value that is incremented for each new commitment transaction which models the rotation of the per_commitment_point. With this approach, the revocation keys can be rotated by modifying the index of the key but they are still regular key pairs. We model the construction that the revocation keys are derived from secrets of both users by changing the conditions in transaction outputs where a revocation key is required: A condition in the output of a

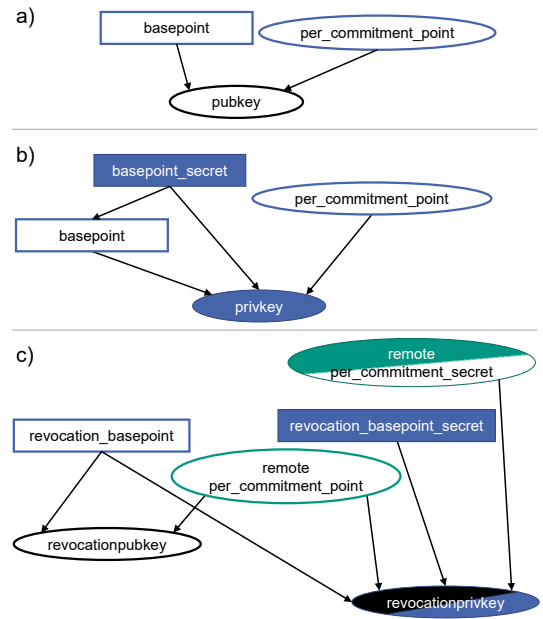


Fig. 9. Derivation of keys in Lightning. Each box shows a key or piece of information that is required to derive a key. From the perspective of a user u , the boxes filled blue are pieces of information that only the user u has. The boxes filled green are pieces of information that only the other user has. The boxes filled white are pieces of information that are shared between both users. The color of the boxes’ borders indicates the user that generates a piece of information. A black border indicates that both users can derive a piece of information. Arrows represent the relation ‘is required to derive’. Rectangular boxes are pieces of information that are constant during the lifetime of the protocol. Ellipses represent pieces of information that are rotated with every commitment transaction. The remote per_commitment_secret and the revocationprivkey are special because the revocationprivkey cannot be generated by neither party until the remote per_commitment_secret is shared which enables the user u to generate the revocationprivkey.

transaction that requires a signature of the revocation key derived from user A’s per_commitment_point and from user B’s revocation_basepoint is modeled as a condition that requires a signature of user A’s revocation key and user B’s user key. This model leads to the following difference in transaction construction between Lightning and the formalization: In Lightning, a transaction that can be published by user A is revocable with a signature that corresponds to the public revocation key derived from user B’s revocation_basepoint and the respective per_commitment_point of user A. In the formalization, a transaction that can be published by user A is revocable with a signature that corresponds to the public revocation key of user A and the public user key of user B. Both approaches have the same effect: Only user B can revoke a transaction published by user A and user B can revoke the transaction only after having received a secret from user A. The message exchange differs in the following way: In Lightning, user A sends the per_commitment_point to user B so that user B can derive the public revocation key. In the formalization, user A sends the public revocation key to user B. When revoking a transaction in Lightning, user A sends the A’s per_commitment_secret to user B. In the formalization,

TABLE VIII
COMPARISON OF KEYS IN LIGHTNING AND THE TLA⁺ FORMALIZATION.

Lightning	Formalization
funding_pubkey, localpubkey, local_delayedpubkey, local_htlcpubkey	User key modeled by one symbolic value per user.
funding_privkey, localprivkey, local_delayedprivkey, local_htlcprivkey	Modeled by the same symbolic value as the public key.
payment_basepoint, delayed_payment_basepoint, htlc_basepoint and associated secrets	Not required because user key is a symbolic value instead being calculated.
revocationpubkey	Modeled as record that contains a base and an index. The base specifies by a symbolic value the user that created the key. The index is a number that can be incremented to create a new key.
revocation_basepoint, revocation_basepoint_secret	Corresponds to the base of the revocation key.
per_commitment_point, per_commitment_secret	Corresponds to the index of the revocation key.

user A revokes a transaction by sending the private revocation key to user B. Table VIII gives an overview of the keys used in Lightning and the corresponding formalization.

7) *Transactions*: The funding transaction contains an output whose amount equals the capacity of the payment channel and which can be spent using signatures for both users' funding_pubkey.

The commitment transaction has one input that references the funding transaction's output. Thus, the commitment transaction must be signed by signatures for both users' funding_pubkey. The outputs of a commitment transaction are as follows:

- to_local: The first output contains the funds of the party who can publish this commitment transaction. The output can be spent using a signature by local_delayedprivkey after a timeout of length to_self_delay or it can be spent using revocationprivkey.
- to_remote: The output that contains funds for the remote party can be spent using remoteprivkey.
- Outgoing HTLCs: For each outgoing HTLC, there exists an output that can be spent either using the revocationprivkey or using remote_htlcprivkey and either using local_htlcprivkey or by providing the HTLC's preimage.
- Incoming HTLCs: For each incoming HTLC, there exists an output that can be spent either using revocationprivkey or using remote_htlcprivkey and either the HTLC's preimage and local_htlcprivkey or after the HTLC's timeout.

HTLC Transactions: The HTLC success transaction has a locktime value of 0 which means that it is immediately valid. The HTLC timeout transaction has a locktime value of cltv_expiry (the HTLC's timeout), i.e. it is valid only after the point in time specified by the HTLC's timeout. An HTLC's transaction input references the respective HTLC output of a commitment transaction. The HTLC transaction must be

signed using remote_htlcprivkey and local_htlcprivkey. For an HTLC success transaction, the input must contain the HTLC's preimage. The output of both HTLC transactions is spendable either by the revocationprivkey or after to_self_delay using local_delayedprivkey.

8) *Formalization of Transactions*: The formalization of transactions follows the UTXO (unspent transaction output) model of Bitcoin: A transaction is a record that contains a set of inputs, a set of outputs and additional data like the transaction's id, and an optional absolute timelock. An output of a transaction is a record that contains an id, an amount, and a set of conditions of which one needs to be fulfilled to spend the output. A condition consists of a type, a set of keys, an optional hash value, an absolute timelock and a relative timelock. The condition can be one of the symbolic values SingleSignature, AllSignatures, SingleSigHashLock, AllSigHashLock. SingleSignature specifies that this output can only be spent by a transaction signed using one of the keys stored in the condition. AllSignatures specifies that a spending transaction must be signed using all of the keys stored in the condition. The types with the suffix 'HashLock' specify the additional constraint that a spending transaction must provide a preimage to the hash value specified in the condition. Additionally, a spending transaction must specify timelocks that have values set to a value that is greater than or equal to the values specified in the condition. The absolute timelock enforces that a spending transaction is only valid if the transaction is published at a point in time that is greater than or equal to the value of the absolute timelock. The relative timelock has the same effect but the transaction must be published at a point in time at which the age of the transaction that the output is contained in is greater than or equal to the relative timelock. An input of a transaction consists of a reference to an output that is being spent by this input, a relative timelock, and witness data which contains of a set of signatures and an optional preimage. Transactions in Bitcoin have ids which are calculated by hashing a transaction. For our model, we require the property that each transaction has a unique id. We model transaction ids by choosing for each transaction an id as an arbitrary integer number that has not yet been used as a transaction id. The variable *cUsedTransactionIds* stores a set of all transaction ids drawn to ensure that each new transaction id is unique. To reduce the number of possible states of the formalization, we specify a unique range of integer numbers for each channel *c*. In the constant *AvailableTransactionIds*, for each channel *c* this set of numbers is stored from which transaction ids created by *PaymentChannelUser* for channel *c* can be drawn.

Lightning requires users to exchange signatures on transactions. Because TLA⁺ does not provide a native way to model signatures, we use an approach that meets the following requirements for signatures: A signature must be bound to the transaction that is signed by the signature so that a user is able to check whether a signature was created for a specific transaction or not. The validity of a signature can be checked using the corresponding public key. However, a signature

must only be creatable using the corresponding private key. To model these requirements in a simple way, we model signatures of transactions by using the whole transaction with the signing key added to the inputs of the transaction. This makes it simple to exchange signed transactions by exchanging the whole transaction and it is trivial to verify the validity of a signature by comparing the signature to the respective public key. A caveat here is that when in Lightning only a signature is exchanged (e.g., in the ‘funding_signed’ and ‘commitment_signed’ messages), both users must be able to create the respective transaction by themselves. To ensure that both users have all the information necessary to create the transactions in the formalization, we model the reception of a message that contains a signed transaction by comparing the received message to a record that contains the expected transaction built from locally available information.

We model the blockchain as a set of transactions. The variable `LedgerTx` is a set that contains the transactions that have been published on the blockchain. For all transactions that are added to `LedgerTx`, an entry in `TxAge` is added that models the time since the transaction was included in the blockchain. If a transaction is published, the transaction’s entry in `TxAge` is initialized to 0. When time is increased, all entries in `TxAge` are increased by the same time difference.

9) *Opening a Payment Channel*: To open a payment channel, the funder of the payment channel sends an `open_channel` message to the user to whom the channel should be opened. The `open_channel` message contains several fields for the parameterization of the channel that we ignore for the TLA^+ formalization (see Table IX). If the receiver wants the channel to be opened, the receiver replies with an ‘accept_channel’ message (see Table X) in which the receiver sends their parameters for the channel to the funder. This includes the basepoints for the keys and the first commitment point to derive the public keys for the first commitment transaction. With this information, the funder creates the funding transaction and the first commitment transaction. The funder sends the ‘funding_created’ message (see Table XI) that contains the funder’s signature of the first commitment transaction and the id of the funding transaction which is required for creating commitment transactions. The other user receives the ‘funding_created’ message, validates and stores the signature. The other user creates the version of the first commitment transaction that the funder can publish and sends a signature for the first commitment transaction to the funder in the ‘funding_signed’ message (see Table XII). Having verified and stored the signature of the first commitment transaction, the funder publishes the funding transaction. After both users have noted that the funding transaction was published and confirmed on the blockchain, both users send each other a ‘channel_ready’ message which indicates that the channel can now be used to process payments. The ‘channel_ready’ message (see Table XIII) contains the second `per_commitment_point` that is required to derive the public keys required to create the second commitment transaction.

TABLE IX
FIELDS OF ‘OPEN_CHANNEL’ MESSAGE.

Variable	Description	Formalization
<code>chain_hash</code>	Hash of the blockchain underlying the payment channel.	Not required because formalization considers only one blockchain.
<code>temporary_channel_id</code>	Temporary id to describe the channel.	Not required because channel is uniquely identified by the pair of the channel parties.
<code>funding_satoshis</code>	Number of satoshis that the funder deposits into the channel.	Field ‘Capacity’
<code>push_msat</code>	Number of millisatoshi that the funder is giving to the other party.	Not included for simplicity.
<code>dust_limit_satoshis</code>	No output is created with an amount less than this value. This prevents outputs from being created that cannot be spent economically because the amount of fees required to spend the output would be higher than the value of the output.	Not required because fees are not modeled.
<code>max_htlc_value_in_flight_msat</code>	Maximum amount of millisatoshi that can be part of HTLCs.	Not included for simplicity.
<code>channel_reserve_satoshis</code>	Amount of satoshis that both users need to keep as their balance and cannot spend to disincentivize cheating attempts.	Not required because the formalization does not consider the game theoretic aspect of whether a user might want to cheat but instead shows that cheating never succeeds.
<code>htlc_minimum_msat</code>	Amount that any HTLC must at least have.	Not required for security properties.
<code>feerate_per_kw</code>	Fee rate for commitment and HTLC transactions.	Not required as blockchain fees are not modeled.
<code>to_self_delay</code>	Number of blocks that an output of the counterparty must be locked until the counterparty can spend it.	Modeled as constant ‘TO_SELF_DELAY’.
<code>max_accepted_htlcs</code>	Maximum number of HTLCs that are accepted at the same time to ensure that messages do not grow too large.	Not required.
<code>funding_pubkey</code>	Public key used for locking the funding output that can only be spent using signatures of keys of both users.	Modeled as the user’s public key.
<code>revocation_basepoint</code>	Point used to calculate revocation keys.	Modeled in form of a public/private revocation key pair.
<code>payment_basepoint</code>	Point used to calculate the payment keys.	Modeled as the user’s public key.
<code>delayed_payment_basepoint</code>	Point used to calculate the delayed payment keys.	Modeled as the user’s public key.
<code>htlc_basepoint</code>	Point used to calculate the HTLC keys.	Modeled as the user’s public key.
<code>first_per_commitment_point</code>	Point to calculate keys for the first commitment transaction.	Only revocation key is rotated. Modeled as key index variable as part of the key.
<code>channel_flags</code>	Flags, e.g. whether this channel is to be announced publicly to the P2P network.	Not required.

TABLE X
FIELDS OF ‘ACCEPT_CHANNEL’ MESSAGE. WITH THE EXCEPTION OF
MINIMUM_DEPTH ALL FIELDS ARE ALSO DESCRIBED IN TABLE IX.

Variable	Description	Formalization
temporary_channel_id	Temporary id to describe the channel.	Not required because channel is uniquely identified by the pair of the channel parties.
dust_limit_satoshis	No output is created with an amount less than this value. This prevents outputs from being created that cannot be spent economically because the amount of fees required to spend the output would be higher than the value of the output.	Not required because fees are not modeled.
max_htlc_value_in_flight_msat	Maximum amount of millisatoshi that can be part of HTLCs.	Not included for simplicity.
channel_reserve_satoshis	Amount of satoshis that both users need to keep as their balance and cannot spend to disincentivize cheating attempts.	Not required because the formalization does not consider the game theoretic aspect of whether a user might want to cheat but instead shows that cheating never succeeds.
htlc_minimum_msat	Amount that any HTLC must at least have.	Not required for security properties.
minimum_depth	Number of blocks created after a transaction has been published until the transaction is considered confirmed.	Not required as transactions are modeled as being finally confirmed immediately.
to_self_delay	Number of blocks that an output of the counterparty must be locked until the counterparty can spend it.	Modeled as constant ‘TO_SELF_DELAY’.
max_accepted_htlcs	Maximum number of HTLCs that are accepted at the same time to ensure that messages do not grow too large.	Not required because message sizes are theoretically not limited.
funding_pubkey	Public key used for locking the funding output that can only be spent using signatures of keys of both users.	Modeled as the user’s public key.
revocation_basepoint	Point used to calculate revocation keys.	Modeled in form of a public/private revocation key pair.
payment_basepoint	Point used to calculate the payment keys.	Modeled as the user’s public key.
delayed_payment_basepoint	Point used to calculate the delayed payment keys.	Modeled as the user’s public key.
htlc_basepoint	Point used to calculate the HTLC keys.	Modeled as the user’s public key.
first_per_commitment_point	Point to calculate keys for the first commitment transaction.	Only revocation key is rotated. Modeled as key index variable as part of the key.

TABLE XI
FIELDS OF ‘FUNDING_CREATED’ MESSAGE.

Variable	Description	Formalization
temporary_channel_id	Temporary id to describe the channel.	Not required because channel is uniquely identified by the pair of the channel parties.
funding_txid	ID of the funding transaction	.
funding_output_index	Output ID for the funding output of the funding transaction.	Not required as funding transaction contains only one output.
signature	Signature of the first commitment transaction	First commitment transaction signed by inserting private key.

TABLE XII
FIELDS OF ‘FUNDING_SIGNED’ MESSAGE.

Variable	Description	Formalization
channel_id	ID of the channel derived from funding transaction	Not required.
signature	Signature of the first commitment transaction	First commitment transaction signed by inserting private key.

10) *Formalization of Opening a Payment Channel:* The module PaymentChannelUser contains the actions to open, update, and close a payment channel. As in the module HTLCUser, the actions of the module PaymentChannelUser are parameterized for a specific user u and a payment channel c of user u . In this section, we use u and c to refer to the user and channel passed as parameters to an action.

To encode in the variables of a user u , in which state of the protocol the user is, the variable $c, uState$ contains a string of characters that describes the state. The state is initialized to ‘init’ and after sending an ‘open_channel’ message, the state advances to ‘open-sent-open-channel’ which means that the user u expects to receive an ‘accept_channel’ message in the next step. Sending and receiving of messages is modeled as described above in Appendix B3. The ‘open_channel’ and ‘accept_channel’ messages contain many fields that are not relevant for the TLA⁺ formalization (see Tables IX and X). On reason for fields not being modeled is that they are used for features that are not included in the TLA⁺ formalization because they are optional features (e.g, ‘push_msat’). Further, the TLA⁺ formalization does not model transaction fees for on-

TABLE XIII
FIELDS OF ‘CHANNEL_READY’ MESSAGE.

Variable	Description	Formalization
channel_id	ID of the channel derived from funding transaction	Not required.
second_per_commitment_point	Point to calculate keys for the second commitment transaction	Only revocation key is rotated. Modeled as key index variable as part of the key.

chain transactions. We leave this aspect out of scope to keep the model and security analysis focused. However, modeling transaction fees could be included in the formalization as part of future work. The basepoints and first commitment point are modeled by sending the public key as explained in Appendix B4.

After having received an ‘accept_channel’ message, a funding user creates the funding transaction. A transaction is created as a record consisting of a set of inputs, a set of outputs, an id and an absolute timelock (see Appendix B4). The id of the funding transaction is chosen as a unique unpredictable identifier. Once created, the funding transaction is stored in the field ‘transactions’ of the variable ‘ $c, uInventory$ ’. The sending of the signature of the first commitment transaction in the ‘funding_created’ message is modeled by the action ‘SendSignedFirstCommitTransaction’. As explained in Appendix B4, sending a signature is modeled by sending the whole transaction with the funder’s key added to the transaction’s input. In the same way, the non-funding user responds with the signed first commitment transaction for the funder in the ‘funding_signed’ message. Once the funder has received the other user’s signature on the first commitment transaction, the funder publishes the funding transaction by adding it to the set of published transactions in the variable ‘LedgerTx’. Together with adding the transaction to LedgerTx, a new entry for the transaction is added to the variable TxAge which maps transaction ids to clocks. A new clock for the published transaction is created and initialized to 0. This clock models the age of the transaction, i.e. how many blocks have been created since the transaction was published. This information is relevant for spending conditions with timelocks.

After the users have noticed that the funding transaction has been published, both users exchange new revocation public keys which models the exchange of per_commitment_points (see Appendix B4). Because each user can send the new revocation public key to the other user after noticing that the funding transaction has been published, the order in which the ‘channel_ready’ messages are sent is not deterministic and a user might first send the ‘channel_ready’ message and then receive the other user’s ‘channels_ready’ message or the other way around. In the formalization, this has the effect that the actions ‘SendNewRevocationKey’ and ‘ReceiveNewRevocationKey’ are enabled in two different states (e.g., ‘open-funding-pub’ and ‘open-new-key-received’) and update the state accordingly (e.g., ‘open-new-key-sent’ and ‘rev-keys-exchanged’). After the new revocation public keys are exchanged, the channel is ready to operate, i.e., to add HTLCs to the commitment transaction.

11) *Updating a Payment Channel:* After a user has sent at least one ‘update_add_htlc’ message (see Table XIV) to inform the other user about an HTLC, the user sends a ‘commitment_signed’ message. The ‘commitment_signed’ message contains a signature for the new commitment transaction in which all outgoing HTLCs for which an ‘update_add_htlc’ message was sent are included and incoming HTLCs that have been fulfilled are not included. The receiving user responds with

TABLE XIV
FIELDS OF ‘COMMITMENT_SIGNED’ MESSAGE.

Variable	Description	Formalization
channel_id	ID of the channel derived from funding transaction	Not required.
signature	Signature of the new commitment transaction	New commitment transaction signed by inserting private key.
num_htlcs	Number of HTLCs in the new commitment transaction	Not required because the field is redundant.
num_htlcs * signature	Signature of each HTLC transaction	HTLC transaction signed by inserting private key.

TABLE XV
FIELDS OF ‘REVOKE_AND_ACK’ MESSAGE.

Variable	Description	Formalization
channel_id	ID of the channel derived from funding transaction	Not required.
per_commitment_secret	Secret for key derivation of keys for the previous commitment transaction.	Only revocation key is rotated. Modeled as sending the private revocation key.
next_per_commitment_point	Point to calculate keys for the new commitment transaction	Only revocation key is rotated. Modeled by sending public revocation key.

a ‘revoke_and_ack’ message (see Table XV) to acknowledge the reception of the new commitment transaction signature and to revoke the old commitment transaction by sending the per_commitment_secret for the old commitment transaction.

12) *Formalization of Updating a Payment Channel:* When the user u is in state ‘rev-keys-exchanged’, the action ‘SendSignedCommitment’ can be enabled if there is at least one HTLC to add or remove and the action ‘ReceiveSignedCommitment’ can be enabled if there is a ‘commitment_signed’ message in the variable $cMessages$ to be received by user u . The actions find HTLCs to be updated by the states of the HTLCs and update the states of the HTLCs according to Fig. 8. The action ‘SendSignedCommitment’ sends the complete new signed commitment transaction and signed HTLC transactions to model the sending of signatures (see Appendix B4). If an outgoing HTLC has timed out, it will not be added by ‘SendSignedCommitment’ and a commitment transaction that commits to an incoming HTLC that has timed out will not be accepted by ‘ReceiveSignedCommitment’.

The formalization keeps track of the *balance* that a user should have in the channel. This value is stored in the variable ‘ $c, uBalance$ ’. During the opening of the payment channel, the value of $c, uBalance$ is set to the channel’s capacity for the funder of the channel and to 0 for the other user. When a user commits to a new HTLC in the action ‘SendSignedCommitment’, the action decrements the user’s balance by the amount of the HTLC. The variable $c, uBalance$ models for each user

the balance that the user is guaranteed to receive as long as the user follows the protocol. The amount of an HTLC is added to a user's balance when the user fulfills the HTLC by sending the preimage to the other user which is modeled by the 'SendHTLCPreimage' action of the module HTLCUser.

Additionally, there is a variable $uChannelBalance$ which models the balance that a user has in all of the user's channels. This variable is part of the security property and updated when a payment is processed. The differences between the variables c , $uBalance$ and $uChannelBalance$ is that c , $uBalance$ represents only the balance that a user has in a channel that is not part of an HTLC while $uChannelBalance$ is the sum of all channels including the amount that is locked in HTLCs.

13) Closing a Payment Channel: There are two ways to close a payment channel: The simplest way is to close the channel by publishing the latest commitment transaction on the blockchain. Both parties can also create a dedicated closing transaction that cannot be revoked and, thus, does not require timeout and which is smaller than a commitment transaction and, thus, costs less fees to publish on the blockchain. For the formalization, we chose to leave this type of closing out of scope because it is an optimization that is useful but not required for a functional and secure protocol. Closing might, however, be dishonest if a party publishes a commitment transaction that is not the latest commitment transaction.

Each party has to watch the blockchain for published commitment transactions and react accordingly. When a channel is closed with an outdated commitment transaction, the honest party has to spend the commitment transaction's revocation outputs. When a channel is closed with a latest commitment transaction that contains HTLCs, these HTLCs need to be resolved on the blockchain. If an incoming HTLC can be fulfilled, an HTLC success transaction must be published and, if an outgoing HTLC is timed out, an HTLC timeout transaction must be published. HTLCs that are not part of the latest commitment transaction, are aborted if they have not been committed, timed out or persisted.

14) Formalization of Closing a Payment Channel: In the formalization, a payment channel can only be closed by publishing a commitment transaction on the blockchain. This can either be done honestly modeled by the action 'CloseChannel' or dishonestly modeled by the action 'Cheat'. We model that a user observes a commitment transaction on the blockchain using different actions depending on whether the other party closed honestly or dishonestly. When the payment channel is closed honestly, the published commitment transaction defines which HTLCs are committed and which are not. The action 'NoteThatOtherPartyClosedHonestly' updates the states of the HTLCs accordingly: The state of HTLCs that were in the process of being committed but are not committed is set to ABORTED. The state of those HTLCs that are committed is set to COMMITTED and the state of HTLCs that were fulfilled and are not in the commitment transaction is set to PERSISTED. When the payment channel is closed dishonestly, the action 'Punish' models that the user u notices the outdated commitment transaction and publishes a transaction that uses

the revocation keys to punish the cheating party.

A challenge for the formalization is that a channel might be closed while the channel is in the process of being updated. A dishonest user might revoke a commitment transaction and publish the revoked commitment transaction. If the honest user observes the published commitment transaction on the blockchain before the honest user receives the revocation key, the honest user treats the publication of the commitment transaction as an honest closing. After having received the revocation key, the honest user has to react to the published commitment transaction as a cheating attempt.

15) Formalization of Messages: The exchange of messages in a channel is modeled by letting the users write messages to a message queue per channel from which each user can read the user's first message. A message is sent by an action that specifies that the channel's message queue is extended by the message that is sent. A message contains a field for the recipient, the sender, the message's type, and the payload. For each type of message, the formalization includes an action that expects such a message and reacts to the message. An action of user u that reads a message contains a condition that checks that the first message sent to user u has the expected type. The action updates the user's state as required and removes the message from the queue. There are two types of variables used for message queues. The global variable Messages is a set that contains all messages for requesting and sending an invoice. There is variable $cMessages$ for each channel c that models a FIFO queue for messages exchanged between the two users of channel c . Because of this approach to model messages, the messages in $cMessages$ are delivered in order and the messages in the global Messages variable can be delivered in an arbitrary order. Messages can be directly received or arbitrarily delayed because after a step in which a message was sent there can be no or an unlimited number of steps that advance time until a step is taken in which the message is received.

16) Formalization of Time Flow: HTLCs as well as commitment transactions use timelocks to enforce that certain actions cannot be done before a certain point in time. The values used in Lightning for timelocks are numbers that indicate a specific height of the Bitcoin blockchain. Because the blockchain grows on average at a constant rate, the height of the blockchain represents a logical time that grows on average linear to the clock time. Because the height of the blockchain represents a logical time, we refer to the height of the blockchain also simply as time. We model the time, i.e., height of the blockchain, as an integer number that can increase in integer steps. The current value of the time is stored in the variable LedgerTime. A step that increases LedgerTime is modeled as an action of the LedgerTime module. A behavior of the system described by the TLA⁺ specification is a sequence of steps of actions of the users (modules PaymentChannelUser and HTLCUser) and of steps that advance the time, i.e., increase the value of LedgerTime. Depending on where in a behavior the steps that advance the time are taken between steps of actions of users, the actions

of users are modeled to happen slowly or quickly relative to the growth of the blockchain.

In some situations, the protocol requires a user to take an action before the blockchain has reached a certain height. An example for such a situation is that, if a user knows the preimage for an HTLC, the user must fulfill the HTLC before the HTLC times out. Consider the following scenario: There is an HTLC from user A to user B with a timeout at time 10. User B knows the preimage and the current time is 8. While the action for fulfilling the HTLC is enabled, the action for increasing the `LedgerTime` is also enabled. It is acceptable that the value of the variable `LedgerTime` is increased to 9. However, if the value of the variable `LedgerTime` was increased to 10 before user B fulfills the HTLC, user B would not have followed the protocol which requires user B to fulfill the HTLC before the HTLC's timeout. Therefore, to model honest behavior of user B, we need to model that user B performs an action before the variable `LedgerTime` reaches the value 10. If the value of the variable `LedgerTime` is 9, the action that increases the value of `LedgerTime` should not be enabled until user B has fulfilled the HTLC. To model this urgency requirement, we let the each module specify a set of points in time called *TimeBounds* that are the heights of the blockchain at which the user needs to perform an action at the latest. The time (resp. height of the blockchain) will not advance further than the minimal height specified by all *TimeBounds*. For the previous example, the value of *TimeBounds* of the module `PaymentChannelUser` would include the HTLC's timelock - 1 for each incoming HTLC that the user has the preimage for and that is not fulfilled. After an action that removes the condition for a time bound has been taken, the height of the blockchain can advance further. We check that no execution of the protocol is stuck because of a time bound but no action is possible and the height of the blockchain cannot advance by using a liveness property that checks that the time finally reaches a specified maximal value.

17) *Liveness of Users*: The Lightning protocol requires that honest users perform certain steps if they can. For example, a user must respond to a 'commitment_signed' message with a 'revoke_and_ack' message. In the TLA⁺ specification, we model these requirements using a weak fairness condition that specifies that, if for an honest user an action is continuously enabled, the user has to eventually take a step of this action. In general, we assume that dishonest users take steps in which they read from the environment but not steps in which they actively change their environment, i.e. dishonest users retrieve messages and read the blockchain but are not required to send messages or publish transactions. However, to a certain degree, we also need to assume liveness for dishonest users. We make the following exceptions: A first exception is that, to achieve progress during channel opening, we specify a weak fairness condition that dishonest users actively participate in the opening of the channel. This simplifies the formalization as we can assume that channels are actually opened. The requirement is not a practical limitation of the adversary as, until a channel has been opened, there is nothing to lose or

gain. If the protocol execution terminates before the funding transaction has been published, the execution had no effect on the blockchain and, thus, on the balances of the users. A second exception is that we specify that, once the other user in the channel has terminated, even dishonest users must publish transactions if they can. This exception simplifies the definition of idealized channels because each user ends in a state in which the user has spent all outputs on the blockchain that the user can spend. This exception is also not a practical limitation of the adversary as the other user who has already terminated will not be able to profit from actions of the adversary anymore. We make a third exception for one specific situation: Assume that an HTLC is part of a commitment transaction that has been honestly published on-chain, i.e. the commitment transaction cannot be revoked. The timeout of the HTLC has already passed. The user for whom the HTLC is incoming is honest but does not have the preimage. The user for whom the HTLC is outgoing is dishonest. Now, the dishonest user could spend the HTLC output and the honest user would note that the HTLC has timedout on-chain. If the dishonest user never spends the HTLC output, the honest user stays in a state in which the HTLC might be resolved in two different ways: Either the dishonest user spends the HTLC output on-chain or the honest user might receive the preimage for the HTLC and spend the HTLC output. Because we specify that users terminate only if they know how all HTLCs have been resolved, this situation prevents honest users from terminating. We decide this situation by specifying that, in this specific situation, a dishonest user publishes a transaction on-chain to timeout the HTLC and retrieve the HTLC's amount. This exception is only required because of the specific termination condition in our formalization. Because the issue is about an output that is expected to be spent by the dishonest user, in practice, this is not a real problem because an honest user retrieves the honest user's balance and it is acceptable if the HTLC is never finally timedout. An alternative to this exception would be to change the protocol so that the honest user marks the HTLC as timedout although the HTLC has not been timedout on-chain.

C. Generalized Time Skip Theorem

Given a real-time specification $Spec_S$, we define a specification $Spec_{\tilde{S}}$ that is implemented by $Spec_S$ and potentially has fewer states. The two specifications differ only in how time is advanced. While in a common real-time specification $Spec_S$ time is advanced by steps of one time unit, the definition of specification $Spec_{\tilde{S}}$ allows time to only advance to points in time at which a new step becomes possible. We say that a step *becomes possible* in a state with time t if the step would not be possible at the directly preceding point in time $t - 1$. The optimized specification does not allow points in time at which every step that is possible could also have been taken at an earlier point in time. For a specification in which at many points in time the same steps are possible, this optimization reduces the state space and, thus, reduces the time required for model checking.

In this section, we define a general real-time specification $Spec_S$ and the corresponding optimized specification $Spec_{\hat{S}}$. We formulate that the original real-time specification implements the optimized specification in Theorem 9 and prove Theorem 9 in Appendix D.

1) *Real-Time Specification $Spec_S$* : The real-time specification $Spec_S$ to be optimized is defined as $Spec_S = Init_S \wedge \Box[Next_S]_{v_S} \wedge Liveness_S$ with the set of variables v_S . The specification $Spec_S$ must be an explicit-time real-time specification with a set of clocks \mathcal{X} and an *AdvanceTime_S* action. We assume that the specification has the following properties: The clocks \mathcal{X} are modeled as variables that are not externally visible. An action may not read a clock and write the clock's value to another variable.⁷ In all initial states, the clocks have the same value. By the *AdvanceTime_S* action, the values of all clocks $x \in \mathcal{X}$ are advanced by 1. This assumption facilitates the proof, however, it is not a real restriction because a specification that allows only +1 advancements of time implements a specification that allows advancements by any natural number. The *AdvanceTime_S* action might be disabled if a time bound is reached. Time bounds are defined for each clock $x \in \mathcal{X}$ by a mapping B^x from states to a set of natural numbers. Time may not advance from a state s if any clock x has a value that is equal to one of the time bounds in the set $B^x(s)$ for clock x and state s . The *Next_S* action of the specification S is a disjunct of an internal next action *NextI* and the *AdvanceTime_S* action: $Next_S = NextI_S \vee AdvanceTime_S$. *Liveness_S* is the conjunction of formulas of the form $WF_{v_S}(A)$ and $SF_{v_S}(A)$ for subactions A of *Next_S*. In this paper, we assume that all values of the clock x are elements of the set \mathbb{N}_0 .

2) *Optimized Specification \hat{S}* : We define the optimized specification \hat{S} with the variables $v_{\hat{S}} = v_S$ as: $Spec_{\hat{S}} = Init_{\hat{S}} \wedge \Box[Next_{\hat{S}}]_{vars_{\hat{S}}} \wedge Liveness_{\hat{S}}$ with $Init_{\hat{S}} = Init_S$ and $Next_{\hat{S}} = AdvanceTime_{\hat{S}} \vee NextI_{\hat{S}}$ and $NextI_{\hat{S}} = NextI_S$ and $Liveness_{\hat{S}} = Liveness_S$. Thus, the only difference between specifications S and \hat{S} is how time is advanced by the *AdvanceTime* actions. To define *AdvanceTime_S*, we introduce the following definitions. We refer to the state space of the optimized specification \hat{S} as $\hat{\Sigma}$.

We define a function \hat{T}_d^x that translates a state s to another time by returning a state s' as a copy of state s with the clock x set to d , i.e. $s'.x = d$.

Definition 3 (\hat{T}_d^x).

$$\begin{aligned} \hat{T}_d^x : \hat{\Sigma} &\rightarrow \hat{\Sigma} \\ s &\mapsto s' \text{ so that } (\forall v \in v_S : s'.v = s.v) \\ &\wedge s'.x = d \end{aligned}$$

Using the definition of \hat{T}_d^x , we can express an assumption on $B^x(s)$. We assume that $B^x(s)$ is independent of the value of the clock x . Formally:

⁷We see no reason why this would be a practical restriction. To measure time differences, new clocks can be created.

Assumption 1.

$$\forall x \in \mathcal{X}, d \in \mathbb{N}_0, s \in \hat{\Sigma} : B^x(s) = B^x(\hat{T}_d^x(s))$$

We extend the definition of \hat{T}_d^x from a function of states to a function $\hat{\mathcal{T}}_d^x$ on behaviors:

Definition 4 ($\hat{\mathcal{T}}_d^x$).

$$\begin{aligned} \hat{\mathcal{T}}_d^x : \hat{\Sigma}^* &\rightarrow \hat{\Sigma}^* \\ \langle \sigma_0, \sigma_1, \sigma_2, \dots \rangle &\mapsto \langle \hat{T}_d^x(\sigma_0), \hat{T}_d^x(\sigma_1), \hat{T}_d^x(\sigma_2), \dots \rangle \end{aligned}$$

Define the set $N_{\hat{S}} \subseteq \hat{\Sigma}^*$ as the set of all behaviors that consist of *NextI_S* or stuttering steps.

Definition 5 ($N_{\hat{S}}$).

$$N_{\hat{S}} = \{ \sigma = \langle \sigma_0, \sigma_1, \dots \rangle \in \hat{\Sigma}^* : \forall i \in \mathbb{N}_0 : \sigma_i \xrightarrow{[NextI_{\hat{S}}]_{v_{\hat{S}}}} \sigma_{i+1} \}$$

and accordingly for specification S

Definition 6 (N_S).

$$N_S = \{ \sigma = \langle \sigma_0, \sigma_1, \dots \rangle \in \Sigma^* : \forall i \in \mathbb{N}_0 : \sigma_i \xrightarrow{[NextI_S]_{v_S}} \sigma_{i+1} \}$$

Using this definition of the set $N_{\hat{S}}$, we can express that a behavior $\sigma \in \hat{\Sigma}^*$ consists of *NextI_S* or stuttering steps by stating that $\sigma \in N_{\hat{S}}$.

We define a *newly possible behavior* as a behavior that contains a step that was not possible if the behavior started at the directly preceding point in time. Formally, a newly possible behavior is a behavior $\sigma \in N_{\hat{S}}$ so that there exists a clock $x \in \mathcal{X}$ so that, while the behavior starts at time $t = \sigma_0.x$, it holds that $\hat{\mathcal{T}}_{t-1}^x(\sigma) \notin N_{\hat{S}}$, i.e., if σ is translated to the directly preceding point in time, then the resulting behavior $\hat{\mathcal{T}}_{t-1}^x(\sigma)$ contains a step that is not allowed by action *NextI_S*.

A newly possible behavior *for state s* , is a newly possible behavior σ that starts in a state created by translating state s to a time $t \in \mathbb{N}$ for a clock $x \in \mathcal{X}$, i.e., $\sigma_0 = \hat{T}_t^x(s)$.

We define the function *ETP^x* (EnablingTimePoints) that maps a state s to all points in time at which a newly possible behavior for state s exists.

Definition 7 (*ETP^x*(s)).

$$\begin{aligned} ETP^x : \Sigma &\rightarrow \mathcal{P}(\mathbb{N}) \\ s &\mapsto \{ t \in \mathbb{N} : \exists \sigma = \langle \sigma_0, \sigma_1, \dots \rangle \in N_S : \\ &\quad \sigma_0 = T_t^x(s) \wedge \mathcal{T}_{t-1}^x(\sigma) \notin N_S \} \end{aligned}$$

We assume a function *relETP^x*(s) that is defined in specification $Spec_S$ and maps a state to a set of natural numbers so that the following conditions are met:

Assumption 2.

$$\begin{aligned} \forall x \in \mathcal{X}, s \in \Sigma : ETP^x(s) &\subseteq relETP^x(s) \\ \wedge \{ b + 1 \mid b \in B^x(s) \} &\subseteq relETP^x(s) \end{aligned}$$

For a state s for each value t in the set *ETP^x*(s), there exists a *NextI_S* step in a behavior that starts at state $\hat{T}_t^x(s)$

(state s where clock x is translated to time t), so that the step is not possible when translating the behavior to time $t - 1$. The set $relETP^x(s)$ contains the points in time that are in $ETP^x(s)$ and may contain additional points.

Using the definition of $relETP^x$, the action $AdvanceTime_{\hat{S}}$ can be defined as an action that sets the new value of each clock $x \in \mathcal{X}$ to a point in time that is in the set $relETP^x$. All other variables are left unchanged:

Definition 8 ($AdvanceTime_{\hat{S}}$). For two states $s, t \in \hat{\Sigma}$, it holds that $s \xrightarrow{AdvanceTime_{\hat{S}}} t$ iff

$$\begin{aligned} & t.time > s.time \\ & \wedge \forall v \in v_{\hat{S}} : t.v = s.v \\ & \wedge \forall x \in \mathcal{X} : t.x = s.x \vee t.x \in relETP^x(s) \\ & \wedge \forall x \in \mathcal{X} : \forall b \in B^x(s) : s.x \leq b \Rightarrow t.x \leq b \end{aligned}$$

With the definition of $AdvanceTime_{\hat{S}}$, the specification $Spec_{\hat{S}}$ is fully defined and we can state the theorem that $Spec_S$ implements $Spec_{\hat{S}}$:

Theorem 9.

$$Spec_S \Rightarrow Spec_{\hat{S}}$$

D. Proof of Generalized Time Skip Theorem

Based on the definitions given above in Appendix C, we prove Theorem 9.

1) *Extended Real-Time Specification $Spec_{S'}$* : Because the proof that specification $Spec_S$ implements specification $Spec_{\hat{S}}$ needs auxiliary variables for defining how a clock is mapped to specification $Spec_S$, we define an extended specification $Spec_{S'}$ that wraps specification $Spec_S$ and adds the auxiliary variables.

Specification $Spec_{S'}$ uses the variables $v_{S'}$ defined as the variables v_S of $Spec_S$ and the auxiliary variables $mappedClock^x$ for each clock $x \in \mathcal{X}$:

$$v_{S'} = v_S \cup \bigcup_{x \in \mathcal{X}} \{mappedClock^x\}$$

Specification $Spec_{S'}$ is defined by $Spec_{S'} = Init_{S'} \wedge \Box[Next_{S'}]_{v_{S'}} \wedge Liveness_{S'}$ where $Init_{S'}$ is defined as:

$$\begin{aligned} Init_{S'} &= Init_S \\ &\wedge \forall x \in \mathcal{X} : mappedClock^x = x \end{aligned}$$

In TLA^+ , $x \mapsto y$ is the notation for a function that maps x to y . Note, that we do not distinguish between a clock x and the value of the clock x to simplify notation.

$Liveness_{S'}$ is defined as $Liveness_S$.

$Next_{S'}$ is defined as $Next_{S'} = AdvanceTime_{S'} \vee NextI_{S'}$ where $NextI_{S'}$ is defined as:

$$\begin{aligned} NextI_{S'} &= NextI_S \\ &\wedge \forall x \in \mathcal{X} : mappedClock'^x = mappedClock^x \end{aligned}$$

$AdvanceTime_{S'}$ is defined as:

$$\begin{aligned} AdvanceTime_{S'} &= \\ &AdvanceTime_S \\ &\wedge \forall x \in \mathcal{X} : mappedClock'^x = \max(\{mappedClock^x\} \\ &\quad \cup \{n \in relETP^x : n \leq x'\}) \end{aligned}$$

$relETP^x$ is used without a state parameter here. The parameter of $relETP^x$ is the ‘current’ state, i.e., the state that is described by the unprimed variables in $AdvanceTime_{S'}$.

Lemma 10. $Spec_S \Rightarrow Spec_{S'}$

PROOF: The lemma follows directly from the definition of $Spec_{S'}$ because the added variables $mappedClock^x$ are auxiliary variables and, in particular, history variables, that do not affect whether a $Next_S$ step is enabled or not.

2) *Proof of Theorem 9*: In the proof, we use the following notation: \bar{F} is formula F in which each clock $x \in \mathcal{X}$ is replaced by the variable $mappedClock^x$. Formally, this can be expressed using the notation F WITH $v_1 \leftarrow e_1, v_2 \leftarrow e_2$ to describe the expression F where variable v_1 is substituted by expression e_1 and variable v_2 is substituted by expression e_2 . With x_1, x_2, x_3, \dots being the clocks in \mathcal{X} :

$$\begin{aligned} \bar{F} &= F \text{ WITH } x_1 \leftarrow mappedClock^{x_1}, \\ &\quad x_2 \leftarrow mappedClock^{x_2}, \\ &\quad x_3 \leftarrow mappedClock^{x_3}, \\ &\quad \dots \end{aligned}$$

Analogously to the definition of \hat{T}_d^x above, we define $T_d'^x$ to be a function that, given a state $s \in \Sigma'$, returns a state s that equals the given state s except that the value of the clock x is set to d .

Definition 11 ($T_d'^x$).

$$\begin{aligned} T_d'^x : \Sigma' &\rightarrow \Sigma' \\ s &\mapsto s' \text{ so that } (\forall v \in v_{S'} : s'.v = s.v) \\ &\quad \wedge s'.x = d \end{aligned}$$

We extend the definition of $T_d'^x$ from a function of states to a function of behaviors:

Definition 12 ($\mathcal{T}_d'^x$).

$$\begin{aligned} \mathcal{T}_d'^x : \Sigma'^* &\rightarrow \Sigma'^* \\ \langle \sigma_0, \sigma_1, \sigma_2, \dots \rangle &\mapsto \langle T_d'^x(\sigma_0), T_d'^x(\sigma_1), T_d'^x(\sigma_2), \dots \rangle \end{aligned}$$

Analogously to $N_{\hat{S}}$, we define $N_{S'} \subseteq \Sigma'^*$ to be the set of all behaviors in specification S' that consist of $NextI_{S'}$ and stuttering steps.

Definition 13 ($N_{S'}$).

$$N_{S'} = \{\sigma = \langle \sigma_0, \sigma_1, \dots \rangle \in \Sigma'^* : \forall i \in \mathbb{N}_0 : \sigma_i \xrightarrow{[NextI_{S'}]_{v_{S'}}} \sigma_{i+1}\}$$

We define $n^x(s)$ to equal the minimal $n \in \mathbb{N}_0$ so that there exists a newly possible behavior for given state s where the

clock x is set to $s.x - n$. We assume that the function \min is defined so that if the parameter of \min is the empty set, \min equals ∞ .

Definition 14 (n^x).

$$n^x(s) := \min(\{n \in \mathbb{N}_0 : \exists \sigma \in N_{S'} : \sigma_0 = T'_{s.x-n}(s) \wedge \mathcal{T}'_{s.x-n-1}(\sigma) \notin N_{S'}\})$$

Lemma 15. Given two states $s, t \in \Sigma'$ and $s \xrightarrow{\text{AdvanceTime}_{S'}} t$, it holds for all $x \in \mathcal{X}$ that:

$$t.\text{mappedClock}^x \geq \max(s.\text{mappedClock}^x, t.x - n^x(t))$$

We write the proof of this lemma in form of a structured proof as introduced in [66]. Given a state $s \in \Sigma'$, we use the notation \ddot{s} to refer to a copy of state s of which the variables mappedClock^x were removed so that $\ddot{s} \in \Sigma = \hat{\Sigma}$.

PROOF: We prove the implication of Lemma 15, by assuming that $s, t \in \Sigma' \wedge s \xrightarrow{\text{AdvanceTime}_{S'}} t$ and proving that $\forall x \in \mathcal{X} : t.\text{mappedClock}^x = \max(s.\text{mappedClock}^x, t.x - n^x(t))$. Let $x \in \mathcal{X}$ be arbitrary but fixed.

$$\langle 1 \rangle 1. t.\text{mappedClock}^x = \max(\{s.\text{mappedClock}^x\} \cup \{n \in \text{relETP}^x(\ddot{s}) : n \leq t.x\})$$

PROOF: By the assumption that $s, t \in \Sigma' \wedge s \xrightarrow{\text{AdvanceTime}_{S'}} t$, by definition of $\text{AdvanceTime}_{S'}$, and by writing the primed variables v' as $t.v$ and the unprimed variables v as $s.v$.

$$\langle 1 \rangle 2. t.\text{mappedClock}^x = \max(s.\text{mappedClock}^x, \max(\{n \in \text{relETP}^x(\ddot{s}) : n \leq t.x\}))$$

PROOF: By $\langle 1 \rangle 1$ and rearranging the values to take the maximum from.

$$\langle 1 \rangle 3. \max(\{n \in \text{relETP}^x(\ddot{s}) : n \leq t.x\}) \geq t.x - n^x(t)$$

PROOF:

$$\langle 2 \rangle 1. \max(\{n \in \text{relETP}^x(\ddot{s}) : n \leq t.x\}) = t.x - \min(\{n \in \mathbb{N}_0 : (t.x - n) \in \text{relETP}^x(\ddot{s})\})$$

PROOF: $\max(\{n \in \text{relETP}^x(\ddot{s}) : n \leq t.x\})$ can be written as $\max(\{n \in \mathbb{N}_0 : n \in \text{relETP}^x(\ddot{s}) \wedge n \leq t.x\})$ because $\text{relETP}^x(\ddot{s}) \subseteq \mathbb{N}_0$. The greatest value n so that $n \leq t.x$ can be written as $t.x - n_0$ for the minimal value n_0 so that $t.x - n_0 \leq t.x$. Then, $\max(\{n \in \text{relETP}^x(\ddot{s}) : n \leq t.x\}) = t.x - \min(\{n \in \mathbb{N}_0 : (t.x - n) \in \text{relETP}^x(\ddot{s}) \wedge (t.x - n) \leq t.x\})$. The statement follows because $t.x - n \leq t.x$ holds for all $n \in \mathbb{N}_0$.

$$\langle 2 \rangle 2. \min(\{n \in \mathbb{N}_0 : (t.x - n) \in \text{relETP}^x(\ddot{s})\}) \geq \min(\{n \in \mathbb{N}_0 : (t.x - n) \in \text{ETP}^x(\ddot{s})\}) = \min(\{n \in \mathbb{N}_0 : \exists \sigma \in N_S : \sigma_0 = \hat{T}_{t.x-n}^x(\ddot{s}) \wedge \hat{\mathcal{T}}_{t.x-n-1}^x(\sigma) \notin N_{S'}\})$$

PROOF: By definition of relETP^x and ETP^x .

$$\langle 2 \rangle 3. \min(\{n \in \mathbb{N}_0 : \exists \sigma \in N_{S'} : \sigma_0 = T'_{t.x-n}(t) \wedge \mathcal{T}'_{t.x-n-1}(\sigma) \notin N_{S'}\}) = n^x(t)$$

PROOF: Definition of n^x .

$$\langle 2 \rangle 4. \forall n \in \mathbb{N}_0 : (\exists \sigma \in N_S : \sigma_0 = \hat{T}_{t.x-n}^x(\ddot{s}) \wedge \hat{\mathcal{T}}_{t.x-n-1}^x(\sigma) \notin N_S \iff \exists \sigma' \in N_{S'} : \sigma'_0 = T'_{t.x-n}(t) \wedge \mathcal{T}'_{t.x-n-1}(\sigma') \notin N_{S'})$$

PROOF: We prove the statement by proving both directions of the implication:

$$\langle 3 \rangle 1. \text{ ASSUME: } n \in \mathbb{N}_0, \sigma \in N_S, \sigma_0 = \hat{T}_{t.x-n}^x(\ddot{s}),$$

$$\hat{\mathcal{T}}_{t.x-n-1}^x(\sigma) \notin N_S$$

$$\text{ PROVE: } \exists \sigma' \in N_{S'} : \sigma'_0 = T'_{t.x-n}(t) \wedge \mathcal{T}'_{t.x-n-1}(\sigma') \notin N_{S'}$$

PROOF: Define a behavior $\sigma' \in N_{S'}$ by adding the variables mappedClock^x to each state of a copy σ' of σ with, for each state σ'_i of the behavior σ' it holds that $\sigma'_i.\text{mappedClock}^x = s.\text{mappedClock}^x$. Because the variables mappedClock^x do not affect the conditions of NextIS' and are not changed by NextIS' and because all other variables are changed by NextIS' as by NextIS , it holds that each step in σ' is a step of NextIS' , which means that $\sigma' \in N_{S'}$.

By definition of $\text{AdvanceTime}_{S'}$, the states s and t differ only by the variables mappedClock^x and x . Because $\hat{T}_{t.x-n}^x(\ddot{s}).x = t.x - n = T'_{t.x-n}(t).x$, it follows that $\sigma'_0 = T'_{t.x-n}(t)$. By the definition of σ' and by the assumption $\hat{\mathcal{T}}_{t.x-n-1}^x(\sigma) \notin N_{\hat{S}}$, it holds that $\mathcal{T}'_{t.x-n-1}(\sigma') \notin N_{S'}$.

$$\langle 3 \rangle 2. \text{ ASSUME: } n \in \mathbb{N}_0, \sigma' \in N_{S'}, \sigma'_0 = T'_{t.x-n}(t), \mathcal{T}'_{t.x-n-1}(\sigma') \notin N_{S'}$$

$$\text{ PROVE: } \exists \hat{\sigma} \in N_{\hat{S}} : \hat{\sigma}_0 = \hat{T}_{t.x-n}^x(\ddot{s}) \wedge \hat{\mathcal{T}}_{t.x-n-1}^x(\hat{\sigma}) \notin N_{\hat{S}}$$

PROOF: Define a behavior $\hat{\sigma} \in N_{\hat{S}}$ by removing the variables mappedClock^x from each state in σ and duplicating the first state, i.e. $\hat{\sigma} = \langle \hat{\sigma}'_0, \hat{\sigma}'_0, \hat{\sigma}'_1, \hat{\sigma}'_2, \dots \rangle$. Because the first step is a stuttering step and $\text{NextIS}' = \text{NextIS}_{\hat{S}}$, it holds that $\hat{\sigma} \in N_{\hat{S}}$.

By definition of $\text{AdvanceTime}_{S'}$, all variables of the states s and t are equal except the variables mappedClock^x and x . It holds that $\hat{T}'_{t.x-n}(t) = \hat{T}_{t.x-n}^x(\ddot{s})$ because in both states the value of clock x is $t.x - n$ and the variables mappedClock^x do not appear in both states. Because $\hat{\sigma}_0 = \hat{\sigma}'_0 = \hat{T}'_{t.x-n}(t)$, it holds that $\hat{\sigma}_0 = \hat{T}_{t.x-n}^x(\ddot{s})$.

By definition of $\hat{\sigma}$ and by the assumption $\mathcal{T}'_{t.x-n-1}(\sigma') \notin N_{S'}$, it holds that $\hat{\mathcal{T}}_{t.x-n-1}^x(\hat{\sigma}) \notin N_{\hat{S}}$.

$$\langle 3 \rangle 3. \text{ Q.E.D.}$$

By $\langle 3 \rangle 1$ and $\langle 3 \rangle 2$.

$$\langle 2 \rangle 5. \text{ Q.E.D.}$$

After rearranging the left part ($\langle 2 \rangle 1$), expanding the definition of relETP ($\langle 2 \rangle 2$), and expanding the definition of $n(t)$ ($\langle 2 \rangle 3$), the conclusion follows by $\langle 2 \rangle 4$.

$$\langle 1 \rangle 4. \text{ Q.E.D.}$$

The lemma follows from $\langle 1 \rangle 2$ by using $\langle 1 \rangle 3$.

Definition 16 ($\text{Inv}_{S'}$). Define an invariant $\text{Inv}_{S'}$ of a state s of specification $\text{Spec}_{S'}$ as:

$$\text{Inv}_{S'}(s) \equiv \forall x \in \mathcal{X} : s.\text{mappedClock}^x \geq s.x - n^x(s)$$

Lemma 17. $\text{Inv}_{S'}$ is an invariant of specification $\text{Spec}_{S'}$:

$$\text{Spec}_{S'} \Rightarrow \Box \text{Inv}_{S'}$$

$$\langle 1 \rangle 1. \text{ Init}_{S'} \Rightarrow \text{Inv}_{S'}$$

Let $x \in \mathcal{X}$ be arbitrary but fixed. By definition of $Init_{S'}$, it holds that $s.mappedClock^x = s.x$. Because it holds that $n^x(s) \geq 0$ by definition of n^x and by definition of min , it holds that $s.mappedClock^x - s.x \geq -n^x(s)$.

$\langle 1 \rangle 2. Inv_{S'} \wedge [Next_{S'}]_{v_{S'}} \Rightarrow Inv'_{S'}$

$\langle 2 \rangle 1. Inv_{S'} \wedge (v'_{S'} = v_{S'}) \Rightarrow Inv'_{S'}$

PROOF: Assume that $Inv_{S'}$ and $v'_{S'} = v_{S'}$. It directly follows that $Inv'_{S'}$.

$\langle 2 \rangle 2. Inv_{S'} \wedge AdvanceTime_{S'} \Rightarrow Inv'_{S'}$

PROOF: We refer to the step's starting state as s and to the ending state as t . Let $x \in \mathcal{X}$ be arbitrary but fixed. Lemma 15 states that it follows from $AdvanceTime_{S'}$ that $t.mappedClock^x \geq \max(s.mappedClock^x, t.x - n^x(t))$. It follows that $t.mappedClock^x \geq t.x - n^x(t)$. This is equal to $Inv_{S'}(t)$ which can be written as $Inv'_{S'}$.

$\langle 2 \rangle 3. Inv_{S'} \wedge Next_{S'} \Rightarrow Inv'_{S'}$

PROOF: We refer to the step's starting state as s and to the ending state as t . Let $x \in \mathcal{X}$ be arbitrary but fixed. We assume that $Inv_{S'} \wedge Next_{S'}$ holds and we prove that $Inv'_{S'}$.

$\langle 3 \rangle 1. n^x(s) \leq n^x(t)$

PROOF: In the case that $n^x(s) = 0$, it follows that $n^x(s) \leq n^x(t)$ because it follows from the definition of $n^x(t)$ that $n^x(t) \geq 0$.

In the following, we assume that $n^x(s) > 0$ which is the only other possible case because $n^x(s) \geq 0$ by definition of $n^x(s)$.

$\langle 4 \rangle 1. \forall n \in \mathbb{N}_0$ it holds that, if $n < n^x(s)$, then $\forall \sigma \in N_{S'} : \sigma_0 = T'_{s.x-n}(s) \Rightarrow T'_{s.x-n-1}(\sigma) \in N_{S'}$

PROOF: Follows from the definition of $n^x(s)$. $n^x(s)$ is defined as $\min(\{n \in \mathbb{N}_0 : f(n)\})$ for a boolean-valued function f . The definition of min implies that $\forall n \in \mathbb{N}_0 : n < n^x(s) \Rightarrow \neg f(n)$.

$\langle 4 \rangle 2. \forall n \in \mathbb{N}_0$ it holds that, if $n < n^x(s)$, then the step from $T'_{s.x-n}(s)$ to $T'_{s.x-n}(t)$ is a $Next_{S'}$ step.

PROOF: We show this by induction over $n \in \mathbb{N}_0$. For $n = 0$, we have to show that the step from $T'_{s.x-0}(s)$ to $T'_{s.x-0}(t)$ is a $Next_{S'}$ step. This holds of $\langle 2 \rangle 3$ because $T'_{s.x}(s) = s$ and the step from s to t is a $Next_{S'}$ step. We assume that if $n < n^x(s)$, then the step from $T'_{s.x-n}(s)$ to $T'_{s.x-n}(t)$ is a $Next_{S'}$ step. We prove that if $n+1 < n^x(s)$, then the step from $T'_{s.x-(n+1)}(s)$ to $T'_{s.x-(n+1)}(t)$ is a $Next_{S'}$ step. Because the step from $T'_{s.x-n}(s)$ to $T'_{s.x-n}(t)$ is a $Next_{S'}$ step, there exists a behavior $\sigma = \langle T'_{s.x-n}(s), T'_{s.x-n}(t), \dots \rangle$. By $\langle 4 \rangle 1$, the behavior $T'_{s.x-(n+1)}(\sigma)$ is a behavior in $N_{S'}$ which means that its first step is a $Next_{S'}$ step from $T'_{s.x-(n+1)}(s)$ to $T'_{s.x-(n+1)}(t)$.

$\langle 4 \rangle 3. \forall n \in \mathbb{N}_0$ it holds that, if $n < n^x(s)$, then $\forall \sigma \in N_{S'} : \sigma_0 = T'_{s.x-n}(t) \Rightarrow T'_{s.x-n-1}(\sigma) \in N_{S'}$

PROOF: By $\langle 4 \rangle 2$, it holds for all $n \in \mathbb{N}_0$ if $n <$

$n^x(s)$, then for every behavior σ^t that starts in state $\sigma_0^t = T'_{s.x-n}(t)$, a behavior σ^s that starts in $\sigma_0^s = T'_{s.x-n}(s)$ can be constructed because the step from $T'_{s.x-n}(s)$ to $T'_{s.x-n}(t)$ is a $Next_{S'}$ step. Because by $\langle 4 \rangle 1$ it holds that $T'_{s.x-n-1}(\sigma^s) \in N_{S'}$, it follows that $T'_{s.x-n-1}(\sigma^t) \in N_{S'}$.

$\langle 4 \rangle 4. \min(\{n \in \mathbb{N}_0 : \exists \sigma \in N_{S'} : \sigma_0 = T'_{t.x-n}(t) \wedge T'_{t.x-n-1}(\sigma) \notin N_{S'}\}) \geq n^x(s)$

PROOF: By $\langle 4 \rangle 3$ and because $s.x = t.x$.

$\langle 4 \rangle 5. \text{Q.E.D.}$

PROOF: By $\langle 4 \rangle 4$ and the definition of $n^x(t) = \min(\{n \in \mathbb{N}_0 : \exists \sigma \in N_{S'} : \sigma_0 = T'_{t.x-n}(t) \wedge T'_{t.x-n-1}(\sigma) \notin N_{S'}\})$.

$\langle 3 \rangle 2. \text{Q.E.D.}$

Because $Next_{S'}$ leaves the variables $mappedClock^x$ and x unchanged, it holds that $t.mappedClock^x = s.mappedClock^x$ and $t.x = s.x$. By $Inv_{S'}$ it follows that $t.mappedClock^x \geq t.x - n^x(s)$ and, by $\langle 3 \rangle 1$, it follows that $t.mappedClock^x \geq t.x - n^x(t)$.

$\langle 2 \rangle 4. \text{Q.E.D.}$

PROOF: Because $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, $\langle 2 \rangle 3$, and $[Next_{S'}]_{v_{S'}} = (AdvanceTime_{S'} \vee Next_{S'}) \vee (v'_{S'} = v_{S'})$.

$\langle 1 \rangle 3. \text{Q.E.D.}$

By $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, and the definition of $Spec_{S'}$.

Lemma 18.

$$Init_{S'} \Rightarrow \overline{Init_{\tilde{S}}}$$

PROOF: Recall that, by definition, $Init_{S'} = Init_{\tilde{S}} \wedge \forall x \in \mathcal{X} : mappedClock^x = x$ and $\overline{Init_{\tilde{S}}} = (Init_{\tilde{S}} \text{ WITH } x_1 \leftarrow mappedClock^{x_1}, x_2 \leftarrow mappedClock^{x_2}, x_3 \leftarrow mappedClock^{x_3}, \dots \text{ and } Init_{\tilde{S}} = Init_{\tilde{S}})$. Because $\forall x \in \mathcal{X} : mappedClock^x = x$, it trivially holds that $Init_{S'} \Rightarrow \overline{Init_{\tilde{S}}}$.

Lemma 19. For every state $s \in \Sigma'$ it holds that if $Inv_{S'}(s)$ then

$$\forall x \in \mathcal{X} : \forall \sigma \in N_{S'} : \sigma_0 = s \Rightarrow T'_{s.mappedClock^x}(\sigma) \in N_{S'}$$

and $Inv_{S'}(T'_{s.mappedClock^x}(s))$.

PROOF: Let $s \in \Sigma'$. We assume that $Inv_{S'}(s)$ holds.

$\langle 1 \rangle 1. \forall x \in \mathcal{X} : \forall \sigma \in N_{S'} : \sigma_0 = s \Rightarrow T'_{s.mappedClock^x}(\sigma) \in N_{S'}$

PROOF: Let $x \in \mathcal{X}$ be arbitrary but fixed.

$\langle 2 \rangle 1. n^x(s) > s.x - s.mappedClock^x - 1$

PROOF: By assumption, $Inv_{S'}(s)$ holds, i.e. $s.mappedClock^x \geq s.x - n^x(s)$. By rearranging, it follows that $n^x(s) \geq s.x - s.mappedClock^x$. Because time is discrete and increases in integer steps, we can replace the \geq by $>$ and it follows that $n^x(s) > s.x - s.mappedClock^x - 1$.

$\langle 2 \rangle 2. \forall n \in \mathbb{N}_0 : n < n^x(s) \Rightarrow \forall \sigma \in N_{S'} : \sigma_0 = s \Rightarrow T'_{s.x-n-1}(\sigma) \in N_{S'}$

PROOF: This follows from the definition of $n^x(s)$. By definition, $n^x(s) = \min(\{n \in \mathbb{N}_0 : \exists \sigma \in N_{S'} : \sigma_0 = T'_{s.x-n}(s) \wedge T'_{s.x-n-1}(\sigma) \notin N_{S'}\})$. In words, $n^x(s)$ is defined as the smallest value by that the value of clock

x in state s must be decreased so that a valid behavior exists that starts in state $T_{s.x-n}^{ix}(s)$ but the behavior $T_{s.x-n}^{ix}(\sigma)$ in which x is one step of time lower is not a valid behavior. Therefore, for all values n smaller than $n^x(s)$, it holds that each behavior that starts in state s can be translated to a valid behavior where the clock x is set to $s.x - n$. Formally, this can be expressed as:

$$\langle 3 \rangle 1. \forall n \in \mathbb{N}_0 : n < n^x(s) \Rightarrow \forall \sigma \in N_{S'} : \sigma_0 = T_{s.x-n}^{ix}(s) \Rightarrow T_{s.x-n-1}^{ix}(\sigma) \in N_{S'}$$

PROOF: From the definition of $n^x(s)$ follows by definition of \min that $\forall n \in \mathbb{N}_0 : n < n^x(s) \Rightarrow \neg(\exists \sigma \in N_{S'} : \sigma_0 = T_{s.x-n}^{ix}(s) \wedge T_{s.x-n-1}^{ix}(\sigma) \notin N_{S'})$. Moving the negation inwards: $\forall n \in \mathbb{N}_0 : n < n^x(s) \Rightarrow \forall \sigma \in N_{S'} : \neg(\sigma_0 = T_{s.x-n}^{ix}(s) \wedge T_{s.x-n-1}^{ix}(\sigma) \notin N_{S'})$

Applying the equivalence $\neg(a \wedge \neg b) = a \Rightarrow b$ from boolean algebra: $\forall n \in \mathbb{N}_0 : n < n^x(s) \Rightarrow \forall \sigma \in N_{S'} : \sigma_0 = T_{s.x-n}^{ix}(s) \Rightarrow T_{s.x-n-1}^{ix}(\sigma) \in N_{S'}$

Informally speaking, $\langle 3 \rangle 1$ states that for all $n < n^x(s)$ all behaviors starting in a state in which the clock x is set to $s.x - n$ can be translated to a valid behavior in which the clock x is set to $s.x - n - 1$. However, the statement to prove requires that all behaviors starting in a state in which the clock x is unchanged, i.e., set to $s.x - 0$, can be translated to a valid behavior in which the clock x is set to $s.x - n - 1$. We use induction over $n \in \mathbb{N}_0$, to show that all the steps reducing the clock x by a single time step can be combined.

$\langle 3 \rangle 2$. Q.E.D.

$$\langle 4 \rangle 1. \text{ If } n = 0, \text{ then } n < n^x(s) \Rightarrow \forall \sigma \in N_{S'} : \sigma_0 = s \Rightarrow T_{s.x-n-1}^{ix}(\sigma) \in N_{S'}$$

PROOF: Follows from $\langle 3 \rangle 1$ by inserting $n = 0$.

Let $n \in \mathbb{N}_0$ be arbitrary but fixed.

$$\langle 4 \rangle 2. \text{ ASSUME: } n < n^x(s) \Rightarrow \forall \sigma \in N_{S'} : \sigma_0 = T_{s.x}^{ix}(s) \Rightarrow T_{s.x-n-1}^{ix}(\sigma) \in N_{S'}$$

$$\text{PROVE: } n + 1 < n^x(s) \Rightarrow \forall \sigma \in N_{S'} : \sigma_0 = T_{s.x}^{ix}(s) \Rightarrow T_{s.x-(n+1)-1}^{ix}(\sigma) \in N_{S'}$$

PROOF:

$$\langle 5 \rangle 1. n + 1 < n^x(s) \Rightarrow \forall \sigma \in N_{S'} : \sigma_0 = T_{s.x}^{ix}(s) \Rightarrow T_{s.x-(n+1)}^{ix}(\sigma) \in N_{S'}$$

It holds that $n + 1 < n^x(s) \Rightarrow n < n^x(s)$ and, by writing $s.x - n - 1$ as $s.x - (n + 1)$, the statement follows from the assumption.

$$\langle 5 \rangle 2. n + 1 < n^x(s) \Rightarrow \forall \tau \in N_{S'} : \tau_0 = T_{s.x-(n+1)}^{ix}(s) \Rightarrow T_{s.x-(n+1)-1}^{ix}(\tau) \in N_{S'}$$

PROOF: By $\langle 3 \rangle 1$ and replacing n by $n + 1$.

$$\langle 5 \rangle 3. n + 1 < n^x(s) \Rightarrow \forall \sigma \in N_{S'} : \sigma_0 = T_{s.x}^{ix}(s) \Rightarrow T_{s.x-(n+1)}^{ix}(\sigma) \in N_{S'} \Rightarrow T_{s.x-(n+1)-1}^{ix}(T_{s.x-(n+1)}^{ix}(\sigma)) \in N_{S'}$$

PROOF: For all $\sigma \in N_{S'}$ with $\sigma_0 = s$ and all $n \in \mathbb{N}_0$ it holds by definition of T^{ix} that the first state of $T_{s.x-(n+1)}^{ix}(\sigma) \in N_{S'}$ is $T_{s.x-(n+1)}^{ix}(s)$. Thus, we can apply $\langle 3 \rangle 1$ with n set to $n + 1$ and the statement follows.

$\langle 5 \rangle 4$. Q.E.D.

$$\text{Because } T_{s.x-(n+1)-1}^{ix}(T_{s.x-(n+1)}^{ix}(\sigma)) = T_{s.x-(n+1)-1}^{ix}(\sigma), \text{ it follows from } \langle 5 \rangle 3 \text{ that } n + 1 < n^x(s) \Rightarrow \forall \sigma \in N_{S'} : \sigma_0 = T_{s.x}^{ix}(s) \Rightarrow T_{s.x-(n+1)-1}^{ix}(\sigma) \in N_{S'}.$$

$\langle 2 \rangle 3$. Q.E.D.

$$\langle 3 \rangle 1. s.x - s.\text{mappedClock}^x - 1 \geq 0$$

PROOF: If $s.x = s.\text{mappedClock}^x$, then Lemma 19 is trivial. Thus, we assume that $s.x \neq s.\text{mappedClock}^x$. By definition of $\text{AdvanceTimes}_{S'}$, it holds that $s.x > s.\text{mappedClock}^x$. It follows that $s.x - s.\text{mappedClock}^x > 0$ and, because time increases in discrete steps, it holds that $s.x - s.\text{mappedClock}^x \geq 1$.

From $\langle 2 \rangle 1$ and $\langle 3 \rangle 1$ it follows that we can insert $s.x - s.\text{mappedClock}^x - 1$ for n in $\langle 2 \rangle 2$. It holds that $\forall \sigma \in N_{S'} : \sigma_0 = s \Rightarrow T_{s.x-(s.x-s.\text{mappedClock}^x-1)-1}^{ix}(\sigma) \in N_{S'}$. This is equal to $\forall \sigma \in N_{S'} : \sigma_0 = s \Rightarrow T_{s.\text{mappedClock}^x}^{ix}(\sigma) \in N_{S'}$.

To simplify notation, we define $w(x, s) = T_{s.\text{mappedClock}^x}^{ix}(s)$ and we omit the parameters, i.e. we write w instead of $w(x, s)$.

$$\langle 1 \rangle 2. \forall x \in \mathcal{X} : w.\text{mappedClock}^x \geq w.x - n^x(w)$$

PROOF: Let $x \in \mathcal{X}$ be an arbitrary but fixed.

$$\langle 2 \rangle 1. n^x(w) \geq 0$$

PROOF: By definition of n^x , n^x maps to values in \mathbb{N}_0 or ∞ .

$$\langle 2 \rangle 2. w.\text{mappedClock}^x = w.x$$

PROOF:

$$\langle 3 \rangle 1. w.x = s.\text{mappedClock}^x$$

PROOF: By definition of T' , T' sets the value of $w.x$ to $s.\text{mappedClock}^x$.

$$\langle 3 \rangle 2. w.\text{mappedClock}^x = s.\text{mappedClock}^x$$

PROOF: By definition of T' , T' sets the value of $w.\text{mappedClock}^x$ to the respective value in s , i.e. $s.\text{mappedClock}^x$.

$$\langle 3 \rangle 3. \text{ Q.E.D.}$$

PROOF: By $\langle 3 \rangle 1$ and $\langle 3 \rangle 2$

$\langle 2 \rangle 3$. Q.E.D.

PROOF: From $\langle 2 \rangle 1$ follows that $0 \geq -n^x(w)$. Adding $w.x$ on both sides of the equation results in $w.x \geq w.x - n^x(w)$. The statement follows by replacing $w.x$ by $w.\text{mappedClock}^x$ because of $\langle 2 \rangle 2$.

Goal: We want to show that the specification S' implements the time-optimized specification \hat{S} . This means we show that the following theorem holds:

Theorem 20. $\text{Spec}_{S'} \Rightarrow \overline{\text{Spec}_{\hat{S}}}$

In the proof, we use the shorthand $L_{S'}$ for $\text{Liveness}_{S'}$ and $L_{\hat{S}}$ for $\text{Liveness}_{\hat{S}}$.

$$\langle 1 \rangle 1. \text{Init}_{S'} \wedge \Box[\text{Next}_{S'}]_{v_{S'}} \wedge L_{S'} \Rightarrow \Box[\text{Next}_{\hat{S}}]_{v_{\hat{S}}} \wedge L_{\hat{S}}$$

$$\langle 2 \rangle 1. \text{Inv}_{S'} \wedge [\text{Next}_{S'}]_{v_{S'}} \Rightarrow \overline{[\text{Next}_{\hat{S}}]_{v_{\hat{S}}}}$$

$$\langle 3 \rangle 1. \text{Inv}_{S'} \wedge (v'_{S'} = v_{S'}) \Rightarrow \overline{\text{Next}_{\hat{S}}} \vee (\overline{v'_{\hat{S}}} = \overline{v_{\hat{S}}})$$

PROOF: Because $\text{Inv}_{S'} \wedge (v'_{S'} = v_{S'}) \Rightarrow (v'_{S'} = v_{S'}) \Rightarrow (\overline{v'_{\hat{S}}} = \overline{v_{\hat{S}}})$.

$$\langle 3 \rangle 2. \text{Inv}_{S'} \wedge \text{Next}_{S'} \Rightarrow \overline{\text{Next}_{\hat{S}}} \vee (\overline{v'_{\hat{S}}} = \overline{v_{\hat{S}}})$$

$$\langle 4 \rangle 1. \text{Inv}_{S'} \wedge \text{AdvanceTimes}_{S'} \Rightarrow \overline{\text{Next}_{\hat{S}}} \vee (\overline{v'_{\hat{S}}} = \overline{v_{\hat{S}}})$$

⟨5⟩1. SUFFICES: ASSUME: $Inv_{S'}$ \wedge
 $AdvanceTime_{S'}$
PROVE: $\overline{Next_{\hat{S}}} \vee (\overline{v'_S} = \overline{v_S})$

PROOF: Obvious.

⟨5⟩2. CASE: $\forall x \in \mathcal{X} : mappedClock^x = mappedClock^x$

PROOF: In this case, $AdvanceTime_{S'}$ does not change the value of the variables $mappedClock^x$. Because, by definition, $AdvanceTime_{S'}$ leaves all variables except $mappedClock^x$ for all $x \in \mathcal{X}$ unchanged, only the clocks are changed. As the values of the clocks do not affect the value of $\overline{v_S}$, it follows that $\overline{v'_S} = \overline{v_S}$.

⟨5⟩3. CASE: $\exists x \in \mathcal{X} : mappedClock^x > mappedClock^x$

PROOF: We refer to the step's starting state as s and to the ending state as t . Let $\mathcal{Y} \subseteq \mathcal{X}$ be the set of clocks y for which $mappedClock'^y > mappedClock^y$. We show that the step from state \bar{s} to state \bar{t} is an $AdvanceTime_{\hat{S}}$ step. To show that, we show that the properties of $AdvanceTime_{\hat{S}}$ defined in Definition 8 are fulfilled.

⟨6⟩1. $\forall y \in \mathcal{Y} : \overline{t.y} > \overline{s.y}$

PROOF: Because $mappedClock'^y > mappedClock^y$ which can also be written as $t.mappedClock^y > s.mappedClock^y$ and $t.mappedClock^y = \overline{t.y}$.

⟨6⟩2. $\forall x \in \mathcal{X} \setminus \mathcal{Y} : \overline{t.x} = \overline{s.x}$

PROOF: By definition of $AdvanceTime_{S'}$, the value of $mappedClock^x$ cannot decrease. Because $x \notin \mathcal{Y}$, it follows that $t.mappedClock^x = s.mappedClock^x$ which equals $\overline{t.x} = \overline{s.x}$.

⟨6⟩3. $\forall y \in \mathcal{Y} : \overline{t.y} \in relETP^y(s)$

PROOF: By definition of $AdvanceTime_{S'}$, it holds that $mappedClock'^y = \max(\{mappedClock^y\} \cup \{t \in relETP^y : t \leq y'\})$. Because $mappedClock'^y > mappedClock^y$, it follows that $mappedClock'^y = \max(\{t \in relETP^y : t \leq y'\})$. We can also write this as $t.mappedClock^y = \max(\{t \in relETP^y(s) : t \leq t.y\}) \in relETP^y(s)$.

⟨6⟩4. $\forall v \in v_{\hat{S}} : \overline{t.v} = \overline{s.v}$

PROOF: For all $v \in v_{\hat{S}}$ it holds that $v' = v$ and, because $\overline{v} = v$ it holds that $\overline{v'} = \overline{v}$.

⟨6⟩5. $\forall x \in \mathcal{X} : \forall b \in B^x(s) : b \geq s.x \Rightarrow \overline{t.x} \leq \overline{b}$

PROOF: Let $x \in \mathcal{X}$ be arbitrary but fixed. Let $b \in B^x(s)$ be arbitrary but fixed.

In this proof, we assume that $\overline{b} \geq s.x$ and we show that it follows that $\overline{t.x} \leq \overline{b}$. The assumption can also be written as $b \geq s.mappedClock^x$.

⟨7⟩1. $b \geq s.x$

PROOF: We prove this by contradiction. Assume that $b < s.x$. Then $(b + 1) \leq s.x$ and $s.mappedClock^x \geq (b + 1) > b$ because $(b + 1) \in relETP^x(s)$. This is a contradiction to $b \geq s.mappedClock^x$.

By Assumption 1, it holds that $\forall d \in \mathbb{N}_0, s \in \hat{\Sigma} : B^x(s) = \overline{B^x(\hat{T}_d^x(s))}$ which means that $B^x(s) = \overline{B^x(s)}$. By $AdvanceTime_{S'}$, it holds for all $b' \in B^x(s)$ that $s.x \leq b' \Rightarrow t.x \leq b'$. Because $B^x(s) = \overline{B^x(s)}$, it also holds for b that $s.x \leq b \Rightarrow t.x \leq b$.

From ⟨7⟩1 it follows that $t.x \leq b$.

By definition, it holds that $t.mappedClock^x \leq t.x$ and, therefore, $t.mappedClock^x \leq b$.

This can also be written as $\overline{t.x} \leq \overline{b}$

⟨6⟩6. Q.E.D.

By ⟨6⟩1, ⟨6⟩3, and ⟨6⟩4 it holds that $AdvanceTime_{\hat{S}}$.

⟨5⟩4. Q.E.D.

PROOF: Because, by definition of $AdvanceTime_{S'}$, it holds that $AdvanceTime_{S'} \Rightarrow mappedClock'^x \geq mappedClock^x$, the steps ⟨5⟩2 and ⟨5⟩3 cover all possible cases. Thereby and by ⟨5⟩1 follows the statement to prove.

⟨4⟩2. $Inv_{S'} \wedge NextI_{S'} \Rightarrow \overline{Next_{\hat{S}}} \vee (\overline{v'_S} = \overline{v_S})$

PROOF: We refer to the step's starting state as s and to the ending state as t . Informally speaking, we show that every $NextI$ step that is allowed in specification S' is also allowed in specification \hat{S} .

⟨5⟩1. SUFFICES: ASSUME: $Inv_{S'} \wedge NextI_{S'}$

PROVE: $\overline{Next_{\hat{S}}} \vee (\overline{v'_S} = \overline{v_S})$

PROOF: Obvious.

⟨5⟩2. $\forall x \in \mathcal{X} : \forall \sigma \in N_{S'} : \sigma_0 = s \Rightarrow \mathcal{T}_{s, mappedClock^x}^{t,x}(\sigma) \in N_{S'}$

PROOF: By Lemma 19 because s is a state of specification $Spec_{S'}$.

⟨5⟩3. $\forall \sigma \in N_{S'} : \sigma_0 = s \Rightarrow \bigcirc_x \in \mathcal{X}(\mathcal{T}_{s, mappedClock^x}^{t,x})(\sigma) \in N_{S'}$

PROOF: By Lemma 19, each behavior in $N_{S'}$ that starts in a state of specification $Spec_{S'}$, is mapped by $\mathcal{T}^{t,x}$ to a behavior in $N_{S'}$ that starts in a state of specification $Spec_{S'}$. Therefore, executing the mappings for each clock consecutively, results in a behavior in $N_{S'}$ that starts in a state of specification $Spec_{S'}$.

⟨5⟩4. $\bigcirc_x \in \mathcal{X}(\mathcal{T}_{s, mappedClock^x}^{t,x})(s) \xrightarrow{NextI_{S'}} \bigcirc_x \in \mathcal{X}(\mathcal{T}_{s, mappedClock^x}^{t,x})(t)$

PROOF: By ⟨5⟩3 and because a behavior σ that starts with the state s followed by state t is in $N_{S'}$.

⟨5⟩5. $\bar{s} \xrightarrow{NextI_{S'}} \bar{t}$

PROOF: By ⟨5⟩4 and because, by definition, $\bar{s} = \bigcirc_x \in \mathcal{X}(\mathcal{T}_{s, mappedClock^x}^{t,x})(s)$.

⟨5⟩6. $NextI_{S'}$

PROOF: By $\langle 5 \rangle 5$ written differently.

$\langle 5 \rangle 7$. Q.E.D.

PROOF: Because $NextI_{S'} = NextI_{\hat{S}}$ and because $Next_{\hat{S}} = NextI_{\hat{S}} \vee AdvanceTime_{\hat{S}}$, it follows from $\langle 5 \rangle 6$ that $Next_{\hat{S}}$ and also that $Next_{\hat{S}} \vee (v'_{\hat{S}} = \overline{v_{\hat{S}}})$.

$\langle 4 \rangle 3$. Q.E.D.

PROOF: By $\langle 4 \rangle 1$ and $\langle 4 \rangle 2$ and because $Next_{S'} = AdvanceTime_{S'} \vee NextI_{S'}$.

$\langle 3 \rangle 3$. Q.E.D.

PROOF: By $\langle 3 \rangle 1$ and $\langle 3 \rangle 2$ and because $[Next_{S'}]_{v_{S'}} = Next_{S'} \vee (v'_{S'} = v_{S'})$.

$\langle 2 \rangle 2$. Q.E.D.

PROOF: By $\langle 2 \rangle 1$ and Lemma 17 and because $L_{S'} = L_{\hat{S}}$.

$$\langle 1 \rangle 2. \frac{Init_{S'} \wedge \square[Next_{S'}]_{v_{S'}} \wedge L_{S'}}{Init_{\hat{S}} \wedge \square[Next_{\hat{S}}]_{v_{\hat{S}}} \wedge L_{\hat{S}}} \Rightarrow$$

PROOF: By $\langle 1 \rangle 1$ and Lemma 18.

$\langle 1 \rangle 3$. Q.E.D.

PROOF: By definitions of $Spec_{S'}$ and $Spec_{\hat{S}}$.

E. Proof for Application of Generalized Time Skip Theorem to Formalization of Lightning

To prove that Theorem 9 can be applied to the formalization of Lightning, we need to prove that, for each clock $x \in \mathcal{X}$, the set $relETP^x$ defined in the specification meets the requirements of Assumption 2 and that the time bounds used in the specification meet the requirements of Assumption 1.

We start by proving that the time bounds specified in SpecificationI.tla meet Assumption 1.

PROOF: $B^{LedgerTime}$ is defined by $TimeBounds$ of $PaymentChannelUser$. As the variable $LedgerTime$ does not occur in the definition of $TimeBounds$, the definition of $TimeBounds$ is independent of $LedgerTime$ and, thus, Assumption 1 holds for $B^{LedgerTime}$.

Similarly, B^{TxAge} is defined by $TxTimeBounds$ of $PaymentChannelUser$. As the variable $TxAge$ does not occur in the definition of $TxTimeBounds$, the definition of $TxTimeBounds$ is independent of $TxAge$ and, thus, Assumption 1 holds for B^{TxAge} .

In conclusion, Assumption 1 holds for all clocks of the specification.

We prove that $relETP^x$ meets Assumption 2. The set $relETP^{LedgerTime}$ is defined as $relETP$ in SpecificationII.

PROOF: First, we prove the second statement:

$$\langle 1 \rangle 1. \forall x \in \mathcal{X}, s \in \Sigma : \{b + 1 \mid b \in B^x(s)\} \subseteq relETP^x(s)$$

PROOF: For $LedgerTime$, the statement directly follows from the definition of $relETP$ in SpecificationII that defines $relETP$ as union of $\{t + 1 : t \in TimeBounds\}$ and another set. For the $TxAge$ clocks, the set of time bounds can only be the empty set or $\{TO_SELF_DELAY - 1\}$. Because the set $relETP^{TxAge}$ contains TO_SELF_DELAY , it holds for all states $s \in \Sigma$ that $\{b + 1 \mid b \in B^{TxAge}(s)\} = \{TO_SELF_DELAY\} \subseteq relETP^{TxAge}(s)$.

$$\langle 1 \rangle 2. \forall x \in \mathcal{X}, s \in \Sigma : ETP^x(s) \subseteq relETP^x(s)$$

PROOF: We have to show that for every $n \in ETP^x(s)$, it holds that $n \in relETP^x(s)$. This means that if there is a newly possible behavior at time n , then time n must be in $relETP^x(s)$.

Our proof strategy is based upon the following observation: For each $n \in ETP^x(s)$, there exists a newly possible behavior σ for clock $x \in \mathcal{X}$ and time $n \in \mathbb{N}$. By definition of $ETP^x(s)$, the behavior σ contains a step $\langle s, t \rangle$ that is not possible at time $n - 1$, i.e., the step $\langle \hat{T}_{n-1}^x(s), \hat{T}_{n-1}^x(t) \rangle$ is not allowed in the specification. In the step $\langle s, t \rangle$, the clock x is unchanged because $\sigma \in N_{\hat{S}}^x$. Define a mapping a from $ETP^x(s)$ to the subactions of the specification, that assigns to each $n \in ETP^x(s)$ a subaction A so that $s \xrightarrow{A} t$ but not $\hat{T}_{n-1}^x(s) \xrightarrow{A} \hat{T}_{n-1}^x(t)$, i.e., a step of action $a(n)$ is not possible if clock x is set to $n - 1$ and becomes possible at time n . For the step $\langle \hat{T}_{n-1}^x(s), \hat{T}_{n-1}^x(t) \rangle$, not to be possible, there must be a condition that contains the clock x because the value of the clock x is the only difference between the states $\hat{T}_{n-1}^x(s)$ and s . Therefore, to find all $n \in ETP^x(s)$, we inspect all subactions of the $NextI$ action of SpecificationI and select the actions that depend on the value of the clock x . We need to prove for each subaction of $NextI$ that all points in time at which a step of this action becomes possible, are included in $relETP^x$. Therefore, our strategy is as follows: We go through all subactions A of $NextI$, we find the conditions under which an A step exists that is enabled at time n but not at time $n - 1$, and verify that in all states s from which a state that meets these conditions can be reached by $NextI$ steps it holds that $n \in relETP^x(s)$.

Actions of module $PaymentChannelUser$: There are nine actions in the module that depend on the value of $LedgerTime$ and three actions that depend on $TxAge$.

$$\langle 2 \rangle 1. \forall s \in \Sigma : \forall n \in ETP^{LedgerTime}(s) : a(n) = SendSignedCommitment \Rightarrow n \in relETP^{LedgerTime}(s)$$

The action $SendSignedCommitment$ commits to new HTLCs by including them in the new commitment transaction that is sent. Outgoing HTLCs are included if they are in state NEW and if their timelock is greater than the current value of $LedgerTime$. For two consecutive points in time $n - 1$ and n , a step of the action $SendSignedCommitment$ is possible at time n but not at time $n - 1$ if a $SendSignedCommitment$ step is allowed at time $n - 1$ that adds an HTLC but the same HTLC could not be added at time n because $n - 1$ is smaller than the HTLC's timelock but n is equal to the HTLC's timelock. It follows that as long as there is an HTLC that is in state NEW, the set $relETP^{LedgerTime}(s)$ must include the HTLC's timelock because at this point in time a new step might become possible that sends a commitment transaction without including this HTLC. This condition is fulfilled by the definition of $TimelockRegions$ of the module $PaymentChannelUser$.

Further, the set $relETP^{LedgerTime}(s)$ must include the

timelock of each HTLC that might come into state NEW. The only way for an HTLC to reach the state NEW is to be created by the action `AddAndSendOutgoingHTLC` in the module `HTLCUser`. This action creates an HTLC for an outgoing payment with the payment's timelock. Thus, the set $relETP^{LedgerTime}(s)$ must include the timelock of each outgoing payment. This condition is fulfilled by the definition of `TimelockRegions` of the module `HTLCUser`. An outgoing payment is created if an incoming HTLC is received that is to be forwarded. The timelock of the newly created payment is taken from the `dataForNextHop` field in the message that was prepared by the original sender of the payment. Thus, the set $relETP^{LedgerTime}(s)$ must include all the timelocks included in the `dataForNextHop` fields for payments that are already in process and, for multi-hop payments that are not yet in process, the set $relETP^{LedgerTime}(s)$ must include the timelocks as they will appear in the `dataForNextHop` field. This condition is fulfilled by the definition of `TimelockRegions` of the module `HTLCUser`. It is possible to predict these timestamps because they are calculated only based on a payment's timelock and the payment's route which are known already in the initial states of the specification.

Given that all these values are included in the set $relETP^{LedgerTime}(s)$, the set $relETP^{LedgerTime}(s)$ contains all points in time at which a `SendSignedCommitment` step becomes possible because the action `SendSignedCommitment` depends on `LedgerTime` only for choosing the HTLCs to add.

$$\langle 2 \rangle 2. \forall s \in \Sigma : \forall n \in ETP^{LedgerTime}(s) : a(n) = \text{ReceiveSignedCommitment} \Rightarrow n \in relETP^{LedgerTime}(s)$$

Analogously to $\langle 2 \rangle 1$, `ReceiveSignedCommitment` accepts a commitment transaction that is being received only if the incoming HTLCs that are committed have a timelock that is greater than `LedgerTime`. An incoming HTLC can be committed only if it is in state NEW. As described above, the set $relETP^{LedgerTime}(s)$ contains an HTLC's timelock already for all HTLCs that are or can come into state NEW. Under this condition, the set $relETP^{LedgerTime}(s)$ contains all points in time at which a `ReceiveSignedCommitment` step becomes possible because the action `ReceiveSignedCommitment` depends on `LedgerTime` only for deciding which HTLCs are allowed to be added in a commitment transaction being received.

$$\langle 2 \rangle 3. \forall s \in \Sigma : \forall n \in ETP^{LedgerTime}(s) : a(n) = \text{ReceiveRevocationKey} \Rightarrow n \in relETP^{LedgerTime}(s)$$

The action `ReceiveRevocationKey` depends on `LedgerTime` for defining which HTLCs are marked as COMMITTED. By definition of `ReceiveRevocationKey`, an incoming HTLC is only committed if it is in state SENT-COMMIT and has a timelock greater than `LedgerTime`. As described above, the set $relETP^{LedgerTime}(s)$ contains an HTLC's timelock already for all HTLCs

that are or can come into state NEW. Additionally, the set $relETP^{LedgerTime}(s)$ must also contain an HTLC's timelock if the HTLC is in state SENT-COMMIT or in one of the states RECV-COMMIT and PENDING-COMMIT which are preceding states of SENT-COMMIT. This condition is fulfilled by the definition of `TimelockRegions` of the module `PaymentChannelUser`. Under this condition, the set $relETP^{LedgerTime}(s)$ contains the timelocks of all HTLCs that are in state SENT-COMMIT or can come into this state.

$$\langle 2 \rangle 4. \forall s \in \Sigma : \forall n \in ETP^{LedgerTime}(s) : a(n) = \text{CloseChannel} \Rightarrow n \in relETP^{LedgerTime}(s)$$

The action `CloseChannel` chooses a transaction for closing based on `LedgerTime`. The channel can always be closed using the latest commitment transaction. The channel can also be closed using a pending new commitment transaction unless there is an HTLC that is in state RECV-COMMIT and has timed out, i.e., the HTLC has a timelock lower than or equal to the value of `LedgerTime`. A step of `CloseChannel` that closes using a pending new commitment transaction becomes possible if a timedout HTLC that is in state RECV-COMMIT is advanced to another state. Because the timelocks of all HTLCs that are in state RECV-COMMIT or any preceding state are included in the set $relETP^{LedgerTime}(s)$, the set $relETP^{LedgerTime}(s)$ contains all points in time at which a step of `CloseChannel` becomes possible.

$$\langle 2 \rangle 5. \forall s \in \Sigma : \forall n \in ETP^{LedgerTime}(s) : a(n) \in \{\text{Cheat}, \text{Punish}, \text{RedeemHTLCAfterClose}\} \Rightarrow n \in relETP^{LedgerTime}(s)$$

The actions `Cheat`, `Punish`, and `RedeemHTLCAfterClose` all use a formula that finds transactions that can be used to spend outputs of published transactions and one of the transactions found is published. This formula depends on `LedgerTime` as well as `TxAge` because outputs might be timelocked and only spendable after an absolute timelock or after a certain time after the transaction containing the output was published. If `LedgerTime` reaches the absolute timelock of an output or `TxAge` reaches the relative timelock of an output, there is a new transaction in the set of publishable transactions and, thus, a step publishing this new transaction becomes possible. Consequently, the set $relETP^{LedgerTime}(s)$ must contain absolute timelocks of outputs that might be spent by such a new transaction. This condition is fulfilled by the definition of `TimelockRegions` of the module `PaymentChannelUser` because `TimelockRegions` contains all absolute timelocks of outputs in the ledger.

To account for the points in time at which an output of a transaction becomes spendable that has not been published yet, the set $relETP^{LedgerTime}(s)$ must also include the absolute timelocks of transactions that are stored in the users' `Inventory` variables. This condition is fulfilled by the definition of `TimelockRegions` of the module `PaymentChannelUser`. Such absolute timelocks in transactions in the users' `Inventory` variables are created

based on the absolute timelocks of uncommitted HTLCs. As shown in the steps above, the timelocks of these HTLCs are included by the definition of TimelockRegions of the module PaymentChannelUser.

$$\langle 2 \rangle 6. \forall s \in \Sigma : \forall n \in ETP^{LedgerTime}(s) : a(n) = \text{NoteThatHTLCFulfilledOnChainByOtherUser} \Rightarrow n \in relETP^{LedgerTime}(s)$$

The action NoteThatHTLCFulfilledOnChain finds an HTLC that has been fulfilled on-chain, adds the preimage to the user's variables, and marks the HTLC as **PERSISTED**. If the HTLC that has been fulfilled has timed out for longer than the grace period, i.e., LedgerTime is greater than the HTLC's timelock plus the duration of the grace period, the HTLC's preimage is added to the set LatePreimages. A step of NoteThatHTLCFulfilledOnChain in which a preimage is added to the set LatePreimages becomes possible if LedgerTime equals the HTLC's timelock + $G + 1$ where G is the duration of the grace period. Therefore, the set $relETP^{LedgerTime}(s)$ must contain for each HTLC that is in a state that can be followed by state **PERSISTED** the value of the HTLC's timelock + $G + 1$. This condition is fulfilled by the definition of TimelockRegions of the module PaymentChannelUser and HTLCUser because for every HTLC that is in a state that can be followed by the state **PERSISTED** or that can be created, these sets contain the value of the HTLC's timelock + $G + 1$.

$$\langle 2 \rangle 7. \forall s \in \Sigma : \forall n \in ETP^{LedgerTime}(s) : a(n) = \text{NoteThatHTLCFulfilledOnChainInOtherChannel} \Rightarrow n \in relETP^{LedgerTime}(s)$$

With respect to the use of *LedgerTime*, the definition of NoteThatHTLCFulfilledOnChainInOtherChannel equals the definition of NoteThatHTLCFulfilledOnChainByOtherUser and, thus, the conclusion follows analogously to $\langle 2 \rangle 6$.

$$\langle 2 \rangle 8. \forall s \in \Sigma : \forall n \in ETP^{TxAge}(s) : a(n) \in \{\text{Cheat}, \text{Punish}, \text{RedeemHTLCAfterClose}\} \Rightarrow n \in relETP^{TxAge}(s)$$

The actions Cheat, Punish, and RedeemHTLCAfterClose all use a formula that finds transactions that can be used to spend outputs of published transactions and one of the transactions found is published. This formula depends on *LedgerTime* as well as *TxAge* because outputs might be timelocked and only spendable after an absolute timelock or after a certain time after the transaction containing the output was published. If *TxAge* reaches the relative timelock of an output, there is a new transaction in the set of publishable transactions and, thus, a step publishing this new transaction becomes possible. Consequently, the set $relETP^{TxAge}(s)$ must contain the relative timelocks of outputs that might be spent by such a new transaction. Because all relative timelocks in the specification equal the constant *TO_SELF_DELAY*, this condition is fulfilled by the definition of AdvanceLedgerTime which assumes that $relETP^{TxAge}(s) = \{\text{TO_SELF_DELAY}\}$.

The module HTLCUser contains five actions that depend

on LedgerTime and no actions that depend on TxAge:

$$\langle 2 \rangle 9. \forall s \in \Sigma : \forall n \in ETP^{LedgerTime}(s) : a(n) = \text{AddAndSendOutgoingHTLC} \Rightarrow n \in relETP^{LedgerTime}(s)$$

The action AddAndSendOutgoingHTLC adds a payment with the lowest timelock of all payments whose timelock is greater than the value of LedgerTime. Thus, a new step of AddAndSendOutgoingHTLC becomes possible if a payment cannot be added anymore because the value of LedgerTime equals the payment's timelock and a payment with the next upcoming timelock becomes the payment with the lowest timelock of all payments whose timelock is greater than the value of LedgerTime. Because the definition of TimelockRegions of the module HTLCUser contains the timelock of each existing or future payment, all points in time at which a new step of AddAndSendOutgoingHTLC becomes possible are included in $relETP^{LedgerTime}(s)$.

$$\langle 2 \rangle 10. \forall s \in \Sigma : \forall n \in ETP^{LedgerTime}(s) : a(n) = \text{SendHTLCPreimage} \Rightarrow n \in relETP^{LedgerTime}(s)$$

The action SendHTLCPreimage depends on LedgerTime and sends the preimage to an HTLC only if the HTLC's timelock plus the constant grace period G is greater than LedgerTime. By this dependency, a SendHTLCPreimage step cannot become possible because, if a SendHTLCPreimage is possible at one point in time, then at any preceding point in time is also smaller than the HTLC's timelock plus the grace period G .

$$\langle 2 \rangle 11. \forall s \in \Sigma : \forall n \in ETP^{LedgerTime}(s) : a(n) = \text{ReceiveHTLCPreimage} \Rightarrow n \in relETP^{LedgerTime}(s)$$

Analogously to the action NoteThatHTLCFulfilledOnChainByOtherUser in the module PaymentChannelUser, the action ReceiveHTLCPreimage adds the preimage of an HTLC to the variable LatePreimages if the HTLC's timelock + G is smaller than LedgerTime. Thus, a new step becomes possible at an HTLC's timelock + $G + 1$. Because the set $relETP^{LedgerTime}(s)$ contains the HTLC's timelock + $G + 1$ for every HTLC that can be fulfilled, this condition is fulfilled.

$$\langle 2 \rangle 12. \forall s \in \Sigma : \forall n \in ETP^{LedgerTime}(s) : a(n) = \text{SendHTLCFail} \Rightarrow n \in relETP^{LedgerTime}(s)$$

The action SendHTLCFail depends on LedgerTime to fail an HTLC that has been committed if the HTLC has timed out. Thus, a SendHTLCFail becomes possible if the value of LedgerTime equals the timelock of an HTLC that is in state **COMMITTED**. As the set $relETP^{LedgerTime}(s)$ contains the timelock of all HTLCs that are committed and can come into state **COMMITTED**, the set $relETP^{LedgerTime}(s)$ contains all points in time at which a step of SendHTLCFail becomes possible.

$$\langle 2 \rangle 13. \forall s \in \Sigma : \forall n \in ETP^{LedgerTime}(s) : a(n) = \text{ReceiveHTLCFail} \Rightarrow n \in relETP^{LedgerTime}(s)$$

The action ReceiveHTLCFail marks an HTLC only as failed if the HTLC's timelock is smaller than or

equal to LedgerTime . This is the same condition as for the action SendHTLCFail and, by $\langle 2 \rangle 12$, the set $\text{relETP}^{\text{LedgerTime}}(s)$ contains all points in time at which a step of ReceiveHTLCFail becomes possible.

The module MultiHopMock contains one action that depends on LedgerTime and no actions that depend on TxAge .

$$\langle 2 \rangle 14. \forall s \in \Sigma : \forall n \in \text{ETP}^{\text{LedgerTime}}(s) : a(n) = \text{ReceivePreimageForIncomingHTLC} \Rightarrow n \in \text{relETP}^{\text{LedgerTime}}(s)$$

The action $\text{ReceivePreimageForIncomingHTLC}$ mocks the reception of a preimage of an HTLC. The action describes steps that add the preimage to the variable LatePreimages and steps that do not add the preimage to LatePreimages . The preimage must be added to LatePreimages if the HTLC's timelock is smaller than or equal to LedgerTime . Thus, at an HTLC's timelock, a step that does not add the preimage to LatePreimages becomes impossible but no new step becomes possible because a step that adds the preimage to LatePreimages is already possible when LedgerTime is smaller than the HTLC's timelock.

F. Proof that Specification (II) refines Specification (III)

Let C be the set of all channels that exist in a specification and U be the set of all users of a specification and U_c the users that are part of channel $c \in C$.

The states of specification (I) are defined by the following variables. The variables that are specific for a user and/or a channel are prefixed with an identifier of the specific user and/or channel. In the proofs, we omit the prefixes when they are irrelevant or it is clear from the context which user or channel is referred to. For each user $u \in U$: $v_u = \{ u\text{PreimageInventory}, u\text{LatePreimages}, u\text{PaymentSecretForPreimage}, u\text{NewPayments}, u\text{Payments}, u\text{ChannelBalance}, u\text{ExtBalance}, u\text{Honest} \}$. For each channel $c \in C$: $v_c = \{ c\text{Messages}, c\text{UsedTransactionIds}, c\text{PendingBalance} \}$. For each channel $c \in C$ and for each user u of channel c : $v_{c,u} = \{ c, u\text{State}, c, u\text{Balance}, c, u\text{Vars}, c, u\text{DetailVars}, c, u\text{Inventory} \}$. Global variables: $v_g = \{ \text{LedgerTime}, \text{TxAge}, \text{Messages}, \text{LedgerTx} \}$.

Specification (II) has the same variables as specification (I). Specification (IIa) has the variables that specification (II) has and additionally a variable $u\text{RequestedInvoices}$ for each user u in which data about the mocked environment is stored. Specifically, it is stored for which payments an invoice was already requested.

The states of specification (III) are defined by the following variables: For each user u : $v_u = \{ u\text{PreimageInventory}, u\text{LatePreimages}, u\text{PaymentSecretForPreimage}, u\text{NewPayments}, u\text{Payments}, u\text{ChannelBalance}, u\text{ExtBalance}, u\text{Honest} \}$. For each channel $c \in C$: $v_c = \{ c\text{Messages}, c\text{PendingBalance} \}$. For each channel c and for each user u of channel c : $v_{c,u} = \{ c, u\text{State}, c, u\text{Balance}, c, u\text{Vars} \}$. Global variables: $v_g = \{ \text{LedgerTime}, \text{Messages} \}$

1) (IIa) \Rightarrow (IIIa):

Definition 21. Define f_c to be the refinement mapping from specification (IIa) to specification (IIIa) for channel c . The refinement mapping f_c is defined using TLA^+ in the file $\text{SpecificationIIatoIIIa.tla}$.

Lemma 22. Specification (IIa) for channel $c \in C$ is a refinement of specification (IIIa) with the refinement mapping f_c .

We check the correctness of this lemma using model checking (see Section VII). We model check specification (IIa) and verify that applying the mapping to each state leads to steps of specification (IIIa).

Lemma 23. The refinement mapping f_c maps steps of $\text{PaymentChannelUser}$ to steps of IdealChannel , steps of HTLCUser to steps of HTLCUser , steps of MultiHopMock to steps of MultiHopMock , and steps of LedgerTime to steps of LedgerTime or stuttering steps.

PROOF: By design, steps of an action A of HTLCUser and MultiHopMock are mapped to steps of the same action A in specification (IIIa). This mapping of steps to steps of the same action is implemented in the refinement mapping by leaving the variables or the fields of variables unchanged by the refinement mapping that are updated by the actions of HTLCUser and MultiHopMock . Because the values of the refinement mapping of all variables except LedgerTime do not depend on LedgerTime , these values are unchanged, if the value of LedgerTime changes. Because the refinement mapping does not change the value of LedgerTime and no action in another module describes a change in LedgerTime , a step of LedgerTime is mapped to a step of LedgerTime . Steps of the module $\text{PaymentChannelUser}$ are mapped to steps of the module IdealChannel .

2) (II) \Rightarrow (III): We prove that specification (II) is a refinement of specification (III) by defining a mapping g from the state space of specification (II) to the state space of specification (III) and by proving that the mapping g is a refinement mapping. The mapping g works as follows: In a first step, the mapping g defines for each channel c in specification (II) a state of specification (IIa). This state of specification (IIa) is mapped to a state of specification (IIIa) using the refinement mapping f_c . The mapping g assigns to a state of specification (II) a state of specification (III) with the channel-specific variables of specification (IIIa) for each channel c and the user-specific and general variables of the given state of specification (II).

An instance of specification (IIa) defines the behavior of one single channel. To define an instance of specification (IIa) for each channel $c \in C$ we use the function m_c to define a state of specification (IIa) based on a state of specification (II). To this end, specification (IIa) has the same global variables v_g used by specification (II) but only the variables v_u for the state of the two participating users of the channel c and the variables $v_{c,u}$ and v_c that are specifically related to the channel c .

Definition 24 (m_c). Define a mapping $m_c : \Sigma_{(II)} \rightarrow \Sigma_{(IIa)}$ so that for a state $s \in \Sigma_{(II)}$, the state $m_c(s)$ assigns to all variables in $\Sigma_{(IIa)}$ the following values:

Each channel-specific variable $v \in v_{c,u} \cup v_c$ in specification (IIa) is assigned the value of the corresponding variable in state s with c prepended to the variable's name. E.g., the variable *ChannelMessages* in specification (IIa) is assigned the value $s.cMessages$.

The user-specific variables $v \in v_u$ in specification (IIa) are assigned the following values. As a helper we define the set of relevant preimages $R_{u,c}$ for each user u of the channel c as the domain of the function *uPaymentSecretForPreimage* combined with the set of all preimages for hashes of HTLCs stored in $c, uVars$.

- *uPreimageInventory* is assigned the value of $s.uPreimageInventory \cap R_{u,c}$.
- *uLatePreimages* is assigned the value of $s.uLatePreimages \cap R_{u,c}$.
- *uPaymentSecretForPreimage* is assigned the value of $s.uPaymentSecretForPreimage$.
- *uNewPayments* is assigned the set of all payments in $s.uNewPayments$ that are initiated by user u , for which the next hop is channel c , or for which there exists an incoming HTLC in $c, uVars$.
- *uChannelBalance* and *uPayments* are assigned the corresponding values in state s .
- *uHonest* and *uExtBalance* are assigned the corresponding values in state s .
- *uRequestedInvoices* is assigned the set of payments of users that are not part of channel c for which an invoice was requested from user u .

The global variables in specification (IIa) are assigned the following values:

- *Messages* is assigned a set that contains all messages in $s.Messages$ for which either the sender or the recipient is a user that is member of channel c .
- *LedgerTx* is the set of all transactions in $s.LedgerTx$ that are related to channel c , i.e., the transaction with the funding output and any directly or indirectly spending transactions.
- *LedgerTime* is assigned the greatest value of the set $cTimelockRegions$ that is lower than $s.LedgerTime$. The set $cTimelockRegions$ consists of the value that was assigned to *LedgerTime* for channel c in the previous step⁸ combined with the union of the *TimelockRegions* sets of *PaymentChannelUser*, *HTLCUser*, and *MultiHopMock* for the users of channel c .

Lemma 25. Given a state of specification (II), then in all channels $c \in C$ created by the mapping m_c it holds that for every step $\langle s, t \rangle$ of an action of an instance of *PaymentChannelUser* or *HTLCUser* for channel c , in all channels $c' \neq c$ the step $\langle m_{c'}(s), m_{c'}(t) \rangle$ is either a stuttering step,

a step of an instance of *MultiHopMock* for channel c' or a or a step of *HTLCUser* for channel c' and the same user as in channel c .

Let $\langle s, t \rangle$ be a step in specification (II) of an action of the modules *PaymentChannelUser* or *HTLCUser* for channel c . The variables $v_{c'}$ and $v_{c',u}$ that are specific to channel c' and the variables of users u who are not part of channel c are unaffected by a step in channel c and, thus, these variables are unchanged in the step $\langle m_{c'}(s), m_{c'}(t) \rangle$.

Variables that are left to discuss because they are possibly changed in the step $\langle m_{c'}(s), m_{c'}(t) \rangle$ are the global variables v_g and the user specific variables v_u for users who are both in channel c' and channel c . These variables are: *Messages*, *LedgerTx*, *LedgerTime*, *TxAge*, *uPreimageInventory*, *uLatePreimages*, *uPaymentSecretForPreimage*, *uChannelBalance*, *uPayments*, *uNewPayments*, *uHonest*, *uExtBalance*, and *uRequestedInvoices*.

The variables *LedgerTx* and *TxAge* are reduced by $m_{c'}$ to values that are specific to channel c' . Thus, a change in channel c does not affect the value of *LedgerTx* and *TxAge* in the step $\langle m_{c'}(s), m_{c'}(t) \rangle$ and these variables are unchanged. The variable *LedgerTime* cannot be changed in step $\langle s, t \rangle$ which is a step of an action of the modules *PaymentChannelUser* or *HTLCUser*. Because the set $cTimelockRegions$ can only become smaller, the value of *LedgerTime* in step $\langle m_{c'}(s), m_{c'}(t) \rangle$ does not change.

The remaining variables are changed by the following actions of the modules *PaymentChannelUser* and *HTLCUser*: *PublishFundingTransaction*, *RedeemHTLCAfterClose*, *NoteThatHTLCFulfilledOnChainByOtherUser*, *NoteThatHTLCFulfilledOnChainInOtherChannel*, *RequestInvoice*, *GenerateAndSendPaymentHash*, *ReceivePaymentHash*, *SendHTLCPreimage*, *ReceiveHTLCPreimage*, *AddAndSendOutgoingHTLC*, and *ReceiveUpdateAddHTLC*. In the following, we prove for each of these actions that a step $\langle s, t \rangle$ of these actions in channel c is a step $\langle m_{c'}(s), m_{c'}(t) \rangle$ in channel c' and either a stuttering step, a step of an action of *MultiHopMock* or of *HTLCUser* for the same user.

$\langle 1 \rangle$ 1. If $\langle s, t \rangle$ is a step of *PublishFundingTransaction* of user u in channel c , then $\langle m_{c'}(s), m_{c'}(t) \rangle$ is either a stuttering step, a step of an action of *MultiHopMock* or of *HTLCUser* for user u .

PROOF: The variables that are changed by *PublishFundingTransaction* and are not specific to channel c are *uExtBalance* and *uChannelBalance*. By definition of m_c , both variables are unchanged in step $\langle m_{c'}(s), m_{c'}(t) \rangle$ if user u is not part of channel c' . In the step $\langle s, t \rangle$ the channel balance of user u is increased by the funding amount which is the user's external balance. This change is described by the action *UserOpensPaymentChannel* of the module *MultiHopMock* which decreases the balance of user u by to zero and adds it to the channel balance of user u stored in the variable *uChannelBalance*. Thus, these variables of user u change in the same way as described by the action *PublishFundingTransaction*. The information that

⁸This value is obtained by adding a helper variable for each channel c to specification (II) that stores the mapped value of *LedgerTime* for channel c .

this balance is stored in an external channel from the view of channel c' is stored by adding a record to the variable $uRequestedInvoices$ that stores the amount that is stored in the other channel. As all other variables remain unchanged, $\langle m_{c'}(s), m_{c'}(t) \rangle$ is a step of `UserOpensPaymentChannel` if user u is part of channel c' and a stuttering step otherwise.

- (1)2. If $\langle s, t \rangle$ is a step of `RedeemHTLCAfterClose` or `SendHTLCPreimage` of user u in channel c , then $\langle m_{c'}(s), m_{c'}(t) \rangle$ is either a stuttering step, a step of an action of `MultiHopMock` or of `HTLCUser` for user u .

PROOF: The variables that are changed by `RedeemHTLCAfterClose` and `SendHTLCPreimage` and are not specific to channel c are $uPayments$ and $uChannelBalance$. By definition of m_c , both variables are unchanged in step $\langle m_{c'}(s), m_{c'}(t) \rangle$ if user u is not part of channel c' . Both variables are only changed if a payment is processed. This can only be the case if u is the receiver of the payment. This change of the variables in step $\langle m_{c'}(s), m_{c'}(t) \rangle$ is described by the action `ProcessOtherPayment` of `MultiHopMock` which describes that any payment of which the user u is the receiver may be processed and the respective balance be received.

- (1)3. If $\langle s, t \rangle$ is a step of `NoteThatHTLCFulfilledOnChainByOtherUser` or `ReceiveHTLCPreimage` of user u in channel c , then $\langle m_{c'}(s), m_{c'}(t) \rangle$ is either a stuttering step, a step of an action of `MultiHopMock` or of `HTLCUser` for user u .

PROOF: The variables that are changed by `NoteThatHTLCFulfilledOnChainByOtherUser` and `ReceiveHTLCPreimage` and are not specific to channel c are $uPreimageInventory$, $uLatePreimages$, $uPayments$ and $uChannelBalance$. By definition of m_c , these variables are unchanged in step $\langle m_{c'}(s), m_{c'}(t) \rangle$ if user u is not part of channel c' . Further, by definition of m_c , the variables $uPreimageInventory$ and $uLatePreimages$ are unchanged if the preimage that is being read from the blockchain is not in the set $R_{u,c'}$. If user u is the receiver of the payment associated with the fulfilled HTLC, then the HTLC is not part of another channel of user u and only the variables $uPayments$ and $uChannelBalance$ are changed. This change in the step $\langle m_{c'}(s), m_{c'}(t) \rangle$ is described by the action `ProcessOtherPayment` of `MultiHopMock` which describes that any payment of which the user u is the sender may be processed and the respective balance be deducted from $uChannelBalance$. If user u is not the receiver of the payment associated with the fulfilled HTLC, then the variables $uPayments$ and $uChannelBalance$ are unchanged in step $\langle s, t \rangle$ and, by definition of $m_{c'}$, they are unchanged in step $\langle m_{c'}(s), m_{c'}(t) \rangle$. The variables $uPreimageInventory$ and $uLatePreimages$ change only in step $\langle m_{c'}(s), m_{c'}(t) \rangle$ if the preimage of the fulfilled HTLC is in $R_{u,c'}$. As channels c' and c are different channels and, if $uPreimageInventory$ is changed by `NoteThatHTLCFulfilledOnChainByOtherUser` or `ReceiveHTLCPreimage`, the received preimage must be for an outgoing HTLC in channel c , the HTLC for which the preimage is received must be

an incoming HTLC in channel c' . Thus, the change to the variables $uPreimageInventory$ and $uLatePreimages$ in step $\langle m_{c'}(s), m_{c'}(t) \rangle$ that a preimage is received is described by the action `ReceivePreimageForIncomingHTLC` of the module `MultiHopMock`.

- (1)4. If $\langle s, t \rangle$ is a step of `NoteThatHTLCFulfilledOnChainInOtherChannel` of user u in channel c , then $\langle m_{c'}(s), m_{c'}(t) \rangle$ is either a stuttering step, a step of an action of `MultiHopMock` or of `HTLCUser` for user u .

PROOF: The variables that are changed by `NoteThatHTLCFulfilledOnChainInOtherChannel` and are not specific to channel c are $uPreimageInventory$ and $uLatePreimages$. By definition of m_c , both variables are unchanged in step $\langle m_{c'}(s), m_{c'}(t) \rangle$ if user u is not part of channel c' . By definition of `NoteThatHTLCFulfilledOnChainInOtherChannel`, the HTLC that is being fulfilled must be an incoming HTLC. Thus, the change to the variables $uPreimageInventory$ and $uLatePreimages$ in step $\langle m_{c'}(s), m_{c'}(t) \rangle$ that a preimage is received is described by the action `ReceivePreimageForIncomingHTLC` of the module `MultiHopMock`.

- (1)5. If $\langle s, t \rangle$ is a step of `RequestInvoice` of user u in channel c , then $\langle m_{c'}(s), m_{c'}(t) \rangle$ is either a stuttering step, a step of an action of `MultiHopMock` or of `HTLCUser` for user u .

PROOF: The action `RequestInvoice` is not specific to a channel but only specific to user u . If user u is in channel c' , step $\langle m_{c'}(s), m_{c'}(t) \rangle$ is also described by `RequestInvoice` of the module `HTLCUser`. If user u is not in channel c' , the global variable `Messages` might be changed in step $\langle m_{c'}(s), m_{c'}(t) \rangle$ if a message is sent from user u to a user of channel c' . Such a step $\langle m_{c'}(s), m_{c'}(t) \rangle$ is described by the action `RequestInvoice` of the module `MultiHopMock`. If no user of channel c' is the recipient of the message sent by u , then, by definition of $m_{c'}$, the variable `Messages` is unchanged in $\langle m_{c'}(s), m_{c'}(t) \rangle$ and $\langle m_{c'}(s), m_{c'}(t) \rangle$ is a stuttering step.

- (1)6. If $\langle s, t \rangle$ is a step of `GenerateAndSendPaymentHash` of user u in channel c , then $\langle m_{c'}(s), m_{c'}(t) \rangle$ is either a stuttering step, a step of an action of `MultiHopMock` or of `HTLCUser` for user u .

PROOF: The action `GenerateAndSendPaymentHash` is not specific to a channel but only specific to user u . If user u is in channel c' , step $\langle m_{c'}(s), m_{c'}(t) \rangle$ is also described by `GenerateAndSendPaymentHash` of the module `HTLCUser`. If user u is not in channel c' , the global variable `Messages` might be changed in step $\langle m_{c'}(s), m_{c'}(t) \rangle$ if user u replies to a message sent from a user of channel c' . Such a step $\langle m_{c'}(s), m_{c'}(t) \rangle$ is described by the action `GenerateAndSendPaymentHash` of the module `MultiHopMock`. The value included in the reply is deterministic as it can be deducted from the payment id for which an invoice is requested. Thus, the action `GenerateAndSendPaymentHash` of the module `MultiHopMock` can define the reply that an actual user would send. If user u replies to a message that was not sent by a user from channel c' , by definition of $m_{c'}$, the

variable Messages is unchanged in $\langle m_{c'}(s), m_{c'}(t) \rangle$ and $\langle m_{c'}(s), m_{c'}(t) \rangle$ is a stuttering step.

- (1)7. If $\langle s, t \rangle$ is a step of ReceivePaymentHash of user u in channel c , then $\langle m_{c'}(s), m_{c'}(t) \rangle$ is either a stuttering step, a step of an action of MultiHopMock or of HTLCUser for user u .

PROOF: The action ReceivePaymentHash is not specific to a channel but only specific to user u . If user u is in channel c' , step $\langle m_{c'}(s), m_{c'}(t) \rangle$ is also described by ReceivePaymentHash of the module HTLCUser. If user u is not in channel c' , the global variable Messages might be changed in step $\langle m_{c'}(s), m_{c'}(t) \rangle$ if user u receives a message sent by a user from channel c' . Such a step $\langle m_{c'}(s), m_{c'}(t) \rangle$ is described by the action ReceivePaymentHash of the module MultiHopMock that describes for the user in channel c' that the received message is removed from the Messages variable. If user u receives a message that was not sent by a user from channel c' , by definition of $m_{c'}$, the variable Messages is unchanged in $\langle m_{c'}(s), m_{c'}(t) \rangle$ and $\langle m_{c'}(s), m_{c'}(t) \rangle$ is a stuttering step.

- (1)8. If $\langle s, t \rangle$ is a step of AddAndSendOutgoingHTLC of user u in channel c , then $\langle m_{c'}(s), m_{c'}(t) \rangle$ is either a stuttering step, a step of an action of MultiHopMock or of HTLCUser for user u .

PROOF: The only variable that is changed by AddAndSendOutgoingHTLC and is not specific to channel c is $uNewPayments$. By definition of m_c , this variable is unchanged in step $\langle m_{c'}(s), m_{c'}(t) \rangle$ if user u is not part of channel c' . The change to the variable $uNewPayments$ is that a payment is removed. This change is described by the action AddOutgoingHTLCToOtherChannel of the module MultiHopMock.

- (1)9. If $\langle s, t \rangle$ is a step of ReceiveUpdateAddHTLC of user u in channel c , then $\langle m_{c'}(s), m_{c'}(t) \rangle$ is either a stuttering step, a step of an action of MultiHopMock or of HTLCUser for user u .

PROOF: The only variable that is changed by ReceiveUpdateAddHTLC and is not specific to channel c is $uNewPayments$. By definition of m_c , this variable is unchanged in step $\langle m_{c'}(s), m_{c'}(t) \rangle$ if user u is not part of channel c' . The change to the variable $uNewPayments$ is that a payment is added that should be forwarded. This change is described by the action AddNewForwardedPayment of the module MultiHopMock. The action AddNewForwardedPayment describes the payments that can be added based on the initial payments of other users and calculates the parameters of the payment to be forwarded based on the position of the channel c' in the path of the payment.

- (1)10. Q.E.D.

PROOF: The steps above discussed all actions that change variables that have an effect on the variables in step $\langle m_{c'}(s), m_{c'}(t) \rangle$. Thus, for all steps $\langle s, t \rangle$ of other actions, the step $\langle m_{c'}(s), m_{c'}(t) \rangle$ is a stuttering step.

Lemma 26. For each channel $c \in C$, the mapping m_c maps a

state of specification (II) to a valid state of specification (IIa) of channel c . Formally: $\forall s \in \Sigma_{(II)}, c \in C : m_c(s) \in \Sigma_{(IIa)}$

We prove the lemma by induction:

Let F_i be the set of initial states of specification S_i .

- (1)1. $\forall s \in F_{(II)}, c \in C : m_c(s) \in F_{(IIa)}$

By definition of the initial states, applying the selection by m_c to an initial state of specification (II) results in an initial state of specification (IIa).

- (1)2. ASSUME: $s \in \Sigma_{(II)}$ and $m_c(s) \in \Sigma_{(IIa)}$ and $\langle s, t \rangle$ is a step of specification (II)

PROVE: $m_c(t) \in \Sigma_{(IIa)}$

Assume that state s is a valid state of specification (II) and $m_c(s)$ is a valid state of specification (IIa). We show that for a step $\langle s, t \rangle$ to a state t of specification (II), it holds that $m_c(t)$ is a valid state of specification (IIa). A step of specification (II) is either a step of a (1) PaymentChannelUser, (2) HTLCUser, (3) LedgerTime, or (4) a final withdraw step.

- (2)1. If $\langle s, t \rangle$ is a step of PaymentChannelUser, then $m_c(t) \in \Sigma_{(IIa)}$

If the step $\langle s, t \rangle$ is a step of PaymentChannelUser, it is either (1.1) a step of an action of PaymentChannelUser for channel c or (1.2) for another channel.

- (3)1. If $\langle s, t \rangle$ is a step of PaymentChannelUser for channel c , then $m_c(t) \in \Sigma_{(IIa)}$

If the step $\langle s, t \rangle$ is a step of an action of PaymentChannelUser for channel c then $\langle m_c(s), m_c(t) \rangle$ is a step of specification (IIa) because specification (IIa) allows exactly the same action of PaymentChannelUser that changes the variables selected by the mapping m_c . Because $\langle m_c(s), m_c(t) \rangle$ is a step of specification (IIa), $m_c(t)$ is a state of specification (IIa).

- (3)2. If $\langle s, t \rangle$ is a step of PaymentChannelUser for a channel other than channel c , then $m_c(t) \in \Sigma_{(IIa)}$

If the step $\langle s, t \rangle$ is a step of an action of PaymentChannelUser for a channel other than channel c , the variables in specification (II) that are specific to channel c do not change between states s and t . However, the variables for the users participating in channel c might change in the step $\langle s, t \rangle$ or global variables might change. For example, a user of channel c might receive a preimage. To account for such changes, specification (IIa) contains the module MultiHopMock. The module MultiHopMock abstracts all actions that can happen in other payment channels. By Lemma 25, it holds that $m_c(t) \in \Sigma_{(IIa)}$ and, thus, $m_c(t) \in \Sigma_{(IIa)}$.

- (3)3. Q.E.D.

By (3)1 and (3)2.

- (2)2. If $\langle s, t \rangle$ is a step of HTLCUser, then $m_c(t) \in \Sigma_{(IIa)}$

A step of HTLCUser is either a step of an action of HTLCUser for channel c or for another channel.

- (3)1. If $\langle s, t \rangle$ is a step of HTLCUser for channel c , then $m_c(t) \in \Sigma_{(IIa)}$

If the step $\langle s, t \rangle$ is a step of an action of HTLCUser for channel c then $\langle m_c(s), m_c(t) \rangle$ is a step of specification

(IIa) because specification (IIa) allows exactly the same action of HTLCUser that changes the variables selected by the mapping m_c . Because $\langle m_c(s), m_c(t) \rangle$ is a step of specification (IIa), $m_c(t)$ is a state of specification (IIa).

⟨3⟩2. If $\langle s, t \rangle$ is a step of HTLCUser for a channel other than channel c , then $m_c(t) \in \Sigma_{(IIa)}$

If the step $\langle s, t \rangle$ is a step of an action of HTLCUser for a channel other than channel c , the variables in specification (II) specifically for channel c do not change between states s and t . However, the variables for the users participating in channel c might change in the step $\langle s, t \rangle$ or global variables might change. As in step ⟨2⟩1, we use the module MultiHopMock to account for this. For every action in HTLCUser that changes a variable that is also used in another channel, there is an action in MultiHopMock that describes the changes to the variables. By Lemma 25, it holds that $m_c(t) \in \Sigma_{(IIa)}$ and, thus, $m_c(t) \in \Sigma_{(IIa)}$.

⟨3⟩3. Q.E.D.

By ⟨3⟩1 and ⟨3⟩2.

⟨2⟩3. If $\langle s, t \rangle$ is a step of LedgerTime, then $m_c(t) \in \Sigma_{(IIa)}$

If the step $\langle s, t \rangle$ is a step of LedgerTime, the only change between states s and t is an increase in LedgerTime. By the definition of m_c , LedgerTime changes in the step $\langle m_c(s), m_c(t) \rangle$ as defined by the module LedgerTime. Thus, the step $\langle m_c(s), m_c(t) \rangle$ is a step of specification (IIa). Thus, $m_c(t) \in \Sigma_{(IIa)}$.

⟨2⟩4. If $\langle s, t \rangle$ is a final withdraw step, then $m_c(t) \in \Sigma_{(IIa)}$

Assume that $\langle s, t \rangle$ is a final withdraw step in specification (II). The final withdraw action is defined in specification (IIa) as in specification (II) with the exception of the new value of $uExtBalance$ for each user u . Thus, we only need to show that the change of $uExtBalance$ in $\langle m_c(s), m_c(t) \rangle$ conforms to the final withdraw action of specification (IIa). The balance of dishonest users does not change in step $\langle s, t \rangle$ which matches the definition of the final withdraw action in specification (IIa). In step $\langle s, t \rangle$, the value of $uExtBalance$ for each honest user u is increased by user u 's on-chain balance. In specification (IIa) the visible on-chain balance might be less than in specification (II) because only the on-chain transaction for channel c are visible. However, because specification (IIa) requires that the value of $uExtBalance$ for an honest user u is at least the user's previous balance increased by the on-chain balance from channel c , the user u 's new external balance can be larger than the balances visible in specification (IIa) and, therefore, $\langle m_c(s), m_c(t) \rangle$ is a step of specification (IIa). It follows that $m_c(t) \in \Sigma_{(IIa)}$.

⟨2⟩5. Q.E.D.

By ⟨2⟩1, ⟨2⟩2, ⟨2⟩3, and ⟨2⟩4 because a step in specification (II) can only be either a step of PaymentChannelUser, HTLCUser, LedgerTime, or a final withdraw step.

⟨1⟩3. Q.E.D.

By induction using ⟨1⟩1 and ⟨1⟩2.

Let F be the set of the refinement mappings f_c for all channels $c \in C$ from specification (IIa) to specification (IIIa) that we defined in the TLA⁺ code of the formalization in *SpecificationIIatoIIIa.tla*.

Definition 27 (*g*). Define the function g from $\Sigma_{(II)}$, the state space of specification (II), to $\Sigma_{(III)}$, the state space of specification (III), as follows. For a state s of specification (II), let $g(s)$ be a state of specification (III) with the following value to variable assignments:

- The global variables v_g are assigned as follows:
 - Messages is assigned $s.Messages$ which equals the value $\bigcup_{c \in C} f_c(m_c(s)).Messages$, the union of all global messages.
 - LedgerTime is assigned the value $s.LedgerTime$ which equals the value $\max_{c \in C} (f_c(m_c(s)).LedgerTime)$.
- The variables v_u that concern one specific user u are assigned the value $s.v$ because these variables are left unchanged by f_c .
- In the set of variables v_c that concern one specific channel $c \in C$, there are the variables $cMessages$ and $cPendingBalance$. These variables are assigned the value $f_c(m_c(s)).ChannelMessages$ and $f_c(m_c(s)).PendingBalance$ respectively.
- For the variables $v \in v_{c,u}$ that concern one specific user u and channel c , the value of the variable v in state $g(s)$ is defined as $f_c(m_c(s)).v$.

Lemma 28. A step of IdealChannel or HTLCUser in specification (IIIa) is also possible if the variables that are reduced by m_c (i.e., $uPreimageInventory$, $uLatePreimages$, $uNewPayments$, Messages) contain additional values that are filtered out by m_c (see Definition 24).

PROOF:

⟨1⟩1. All conditions on the first state of a step of IdealChannel or HTLCUser are valid even if the variables that are reduced by m_c contain additional values that are filtered out by m_c .

PROOF: It can be checked for every action of IdealChannel and HTLCUser and every variable that is reduced by m_c that this property holds. For this proof, we discuss all relevant cases. The remaining cases are either trivial or analogous to the discussed cases.

The action RequestInvoice of HTLCUser requests an invoice only if there are no payments with lower timelock in $uNewPayments$ that are initiated by the user u . Because m_c retains all payments that are initiated by the user u in $uNewPayments$, the additional payments that are filtered out by m_c are not relevant for this condition.

The action GenerateAndSendPaymentHash of HTLCUser adds a preimage for the requested payment to $uPreimageInventory$ and a payment secret to $uPaymentSecretForPreimage$. The action contains a condition that the generated preimage may not already be in $uPreimageInventory$. Because the value chosen for the

preimage in the formalization depends on and is unique for the payment that the preimage is for, if the preimage has already been generated then it must have been generated by this action and must have been added to the domain of $u\text{PaymentSecretForPreimage}$ and, therefore, it is selected by m_c .

The action $\text{AddAndSendOutgoingHTLC}$ selects a payment that has the lowest timelock of all payments for which the next hop is this channel. Because m_c retains all payments in $u\text{NewPayments}$ for which the next hop is this channel, additional payments that are not filtered out by m_c do not affect this condition.

In IdealChannel , some actions have checks whether the preimage of an HTLC is in $u\text{PreimageInventory}$. Because these checks are only for preimages for HTLCs of the users and m_c retains all preimages for which an HTLC exists, these checks return the same result if $u\text{PreimageInventory}$ contains more values.

- ⟨1⟩2. All conditions on the second state of a step of IdealChannel or HTLCUser are valid even if the variables that are reduced by m_c contain additional values that are filtered out by m_c .

PROOF: The values of the concerned variables in the second state of a step are always described by the actions of IdealChannel and HTLCUser by describing a change and not directly the new value. More concretely, the actions define that a value is added or removed to the set and all remaining values of the set remain unchanged. Thus, a step of IdealChannel or HTLCUser is still valid if the concerned variables contain additional values.

- ⟨1⟩3. Q.E.D.

PROOF: Because the conditions on the first and the second state are valid, a step of IdealChannel or HTLCUser is also a valid step if the variables that are reduced by m_c contain additional values that are filtered out by m_c .

Theorem 29. *The function g is a refinement mapping from specification (II) to specification (III).*

- ⟨1⟩1. For all $s \in \Sigma_{(II)} : \Pi(g(s)) = \Pi(s)$

PROOF: The externally visible variables are for all users $u \in U$: $u\text{Payments}$, $u\text{ExtBalance}$, $u\text{ChannelBalance}$, and $u\text{Honest}$. These variables are by definition unchanged by g .

- ⟨1⟩2. For all initial states s of specification (II), $g(s)$ is an initial state of specification (III)

PROOF: All variables in state s are mapped by g either by directly assigning the value of the variable in state s or by applying functions $f_c \in F$. By definition of f_c , f_c does not change values in the initial states. By definition of Init of specifications (II) and (III), it follows that all variables of a state of specification (II) are mapped to values of an initial state of specification (III).

- ⟨1⟩3. For all steps $\langle s, t \rangle$ of specification (II), $\langle g(s), g(t) \rangle$ is a step of specification (III).

PROOF:

A step of specification (II) can be one of the following.

- ⟨2⟩1. CASE: $\langle s, t \rangle$ is a step of an action of the module

LedgerTime in specification (II).

PROVE: $\langle g(s), g(t) \rangle$ is a step of the module LedgerTime in specification (III).

The only variable that is updated in step is the variable LedgerTime . Because the way that the refinement mappings f_c update variables does not depend on the value of LedgerTime , all variables other than LedgerTime are unchanged in step $\langle g(s), g(t) \rangle$. It follows that $\langle g(s), g(t) \rangle$ is a step of the module LedgerTime in specification (III) because all variables except LedgerTime are unchanged and $g(s).\text{LedgerTime} = g(t).\text{LedgerTime}$ and specification (III) allows LedgerTime to increase by at least the values that are allowed in specification (II) because specification (III) is a regular (unoptimized) real-time specification.

- ⟨2⟩2. CASE: $\langle s, t \rangle$ is a step of an action of the module HTLCUser or $\text{PaymentChannelUser}$ in specification (II).

PROVE: $\langle g(s), g(t) \rangle$ is a step of HTLCUser or IdealChannel in specification (III).

Let $c \in C$ be the channel and u the user for which the step $\langle s, t \rangle$ is a step of an action of HTLCUser or $\text{PaymentChannelUser}$. By the proof of Lemma 26, the step $\langle m_c(s), m_c(t) \rangle$ is a step of an action of HTLCUser or $\text{PaymentChannelUser}$ for user u in the instance of specification (IIa) that is described by the mapping m_c . By Lemma 25, the step $\langle m_{c'}(s), m_{c'}(t) \rangle$ is a step of HTLCUser , MultiHopMock or a stuttering step in the instance of specification (IIa) for every channel $c' \neq c$ that is described by the mapping $m_{c'}$. By Lemma 22, the step $\langle f_c(m_c(s)), f_c(m_c(t)) \rangle$ is a step of HTLCUser or IdealChannel in specification (IIIa) and the step $\langle f_{c'}(m_{c'}(s)), f_{c'}(m_{c'}(t)) \rangle$ is a step of HTLCUser , MultiHopMock or a stuttering step in specification (IIIa). The channel-specific variables of all channels $c' \neq c$ and the user-specific variables of all users $u' \neq u$ are unchanged in the step $\langle s, t \rangle$ and, by definition of g , these variables are also unchanged in step $\langle g(s), g(t) \rangle$. Therefore, only the global variables, user-specific variables for user u and channel-specific variables for channel c might be changed in step $\langle g(s), g(t) \rangle$. Because the step $\langle f_c(m_c(s)), f_c(m_c(t)) \rangle$ is a step of HTLCUser or IdealChannel in specification (IIIa), by Lemma 28 and the definition of g , the step $\langle g(s), g(t) \rangle$ is a step of HTLCUser or IdealChannel in specification (III).

- ⟨1⟩4. Q.E.D.

PROOF: By ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, and the definition of refinement mappings and because specification (III) contains the same fairness condition as specification (II) that only behaviors are valid that end in a state in which all users have withdrawn their balance.

G. Proof of Application of Generalized Time Skip Theorem to Specification (III)

Analogously to Appendix E, to prove that Theorem 9 can be applied to specification (III), we need to prove that, for each

clock $x \in \mathcal{X}$, the set $relETP^x$ defined in the specification meets the requirements of Assumption 2 and that the time bounds used in the specification meet the requirements of Assumption 1. In contrast to specification (I), specification (III) contains only one clock which is $LedgerTime$.

We start by proving that the time bounds specified in SpecificationIII.tla meet Assumption 1.

PROOF: $B^{LedgerTime}$ is defined by $TimeBounds$ of $IdealChannel$. As the variable $LedgerTime$ does not occur in the definition of $TimeBounds$, the definition of $TimeBounds$ is independent of $LedgerTime$ and, thus, Assumption 1 holds for $B^{LedgerTime}$.

We prove that $relETP^x$ meets Assumption 2. The set $relETP^{LedgerTime}$ is defined as $relETP$ in SpecificationIV.

PROOF: First, we prove the second statement:

$\langle 1 \rangle 1. \forall x \in \mathcal{X}, s \in \Sigma : \{b + 1 \mid b \in B^x(s)\} \subseteq relETP^x(s)$

PROOF: The statement directly follows from the definition of $relETP$ in SpecificationIIa that defines $relETP$ as union of $\{t + 1 : t \in TimeBounds\}$ and another set.

$\langle 1 \rangle 2. \forall x \in \mathcal{X}, s \in \Sigma : ETP^x(s) \subseteq relETP^x(s)$

PROOF: We have to show that for every $n \in ETP^x(s)$, it holds that $n \in relETP^x(s)$. This means that if there is a newly possible behavior at time n , then time n must be in $relETP^x(s)$.

We use the same proof strategy as Appendix E. We go through all subactions A of $NextI$, we find the conditions under which an A step exists that is enabled at time n but not at time $n - 1$, and verify that in all states s from which a state that meets these conditions can be reached by $NextI$ steps it holds that $n \in relETP^x(s)$.

Define a mapping a from $ETP^x(s)$ to the subactions of the specification, that assigns to each $n \in ETP^x(s)$ a subaction A so that $s \xrightarrow{A} t$ but not $\hat{T}_{n-1}^x(s) \xrightarrow{A} \hat{T}_{n-1}^x(t)$, i.e., a step of action $a(n)$ is not possible if clock x is set to $n - 1$ and becomes possible at time n .

The module $HTLCUser$ contains five actions that depend on $LedgerTime$ that have already been discussed in Appendix E. We do not repeat the proofs for these actions here.

The module $IdealChannel$ contains seven actions that depend on $LedgerTime$:

$\langle 2 \rangle 1. \forall s \in \Sigma : \forall n \in ETP^{LedgerTime}(s) : a(n) = UpdatePaymentChannel \Rightarrow n \in relETP^{LedgerTime}(s)$

PROOF: The action $UpdatePaymentChannel$ depends on the value of $LedgerTime$ to choose the HTLCs to add and to remove. For this, the condition that is used is the condition that an HTLC's timelock is greater than $LedgerTime$. Thus, the set $relETP^{LedgerTime}(s)$ must contain an HTLC's timelock because at the timelock of an HTLC a new step becomes possible that does not add a specific HTLC or that removes an HTLC that has timedout. This is fulfilled by the definition of $TimelockRegions$ of the module $IdealChannel$ that is

included by $relETP^{LedgerTime}(s)$.

The action $UpdatePaymentChannel$ also contains a condition that under certain conditions the value of $LedgerTime$ is smaller than an HTLC's absolute timelock plus the grade period G . For larger values of $LedgerTime$ steps become impossible but this condition does not allow for new steps to become possible.

$\langle 2 \rangle 2. \forall s \in \Sigma : \forall n \in ETP^{LedgerTime}(s) : a(n) = SetOnChainHTLCsAndCheater \Rightarrow n \in relETP^{LedgerTime}(s)$

PROOF: As $UpdatePaymentChannel$, the action $SetOnChainHTLCsAndCheater$ contains a condition that under certain conditions the value of $LedgerTime$ is smaller than an HTLC's absolute timelock plus the grade period G . For larger values of $LedgerTime$ steps become impossible but this condition does not allow for new steps to become possible.

Another condition requires a user to have at least the balance of incoming HTLCs that cannot be persisted because the user was dishonest and the value of $LedgerTime$ is larger than or equal to the HTLC's timelock + G . This condition might lead to steps becoming impossible but there are no steps of $SetOnChainHTLCsAndCheater$ that can become possible.

$\langle 2 \rangle 3. \forall s \in \Sigma : \forall n \in ETP^{LedgerTime}(s) : a(n) = FulfillIncomingHTLCsOnChain \Rightarrow n \in relETP^{LedgerTime}(s)$

PROOF: The action $FulfillIncomingHTLCsOnChain$ can fulfill an HTLC on-chain as long as the HTLC's timelock + G is greater than $LedgerTime$. Because the action chooses a subset of fulfillable HTLCs as the HTLCs to fulfill, if the value of $LedgerTime$ reaches an HTLC's timelock + G no new step becomes possible but only all steps that include fulfilling this HTLC become impossible.

Additionally, the action has the same condition for the balance of users as $SetOnChainHTLCsAndCheater$ that also does not lead to steps becoming possible.

$\langle 2 \rangle 4. \forall s \in \Sigma : \forall n \in ETP^{LedgerTime}(s) : a(n) = NoteFulfilledHTLCsOnChain \Rightarrow n \in relETP^{LedgerTime}(s)$

PROOF: The action $NoteFulfilledHTLCsOnChain$ can note a fulfilled HTLC on-chain as long as the HTLC's timelock + G is greater than $LedgerTime$. Because the action chooses a subset of fulfillable HTLCs as the HTLCs to fulfill, if the value of $LedgerTime$ reaches an HTLC's timelock + G no new step becomes possible but only all steps that include fulfilling this HTLC become impossible.

If the preimage for an HTLC is learned when the value of $LedgerTime$ is greater than the HTLC's timelock + G , the preimage is added to the set $uLatePreimages$. Thus, for each HTLC that can be fulfilled, the set $relETP^{LedgerTime}(s)$ must include the HTLC's timelock + $G + 1$. This is fulfilled by the definition of $TimelockRegions$ of the module $IdealChannel$ that is included by

$$\begin{aligned} & relETP^{LedgerTime}(s). \\ \langle 2 \rangle 5. \quad \forall s \in \Sigma & : \quad \forall n \in ETP^{LedgerTime}(s) & : \\ & a(n) = CommitHTLCsOnChain \Rightarrow \\ & n \in relETP^{LedgerTime}(s) \end{aligned}$$

PROOF: The action CommitHTLCsOnChain can persist an HTLC on-chain as long as the HTLC's timelock + G is greater than LedgerTime. Because the action chooses a subset of persistable HTLCs as the HTLCs to persist, if the value of LedgerTime reaches an HTLC's timelock + G no new step becomes possible but only all steps that include persisting this HTLC become impossible.

The action CommitHTLCsOnChain also contains a condition that under certain conditions the value of LedgerTime is smaller than an HTLC's absolute timelock plus the grade period G . For larger values of LedgerTime steps become impossible but this condition does not allow for new steps to become possible.

If the preimage for an HTLC is persisted when the value of LedgerTime is greater than the HTLC's timelock + G, the preimage is added to the set $uLatePreimages$ of the user who learns the preimage. Thus, for each HTLC that can be persisted, the set $relETP^{LedgerTime}(s)$ must include the HTLC's timelock + G + 1. This is fulfilled by the definition of TimelockRegions of the module IdealChannel that is included by $relETP^{LedgerTime}(s)$.

Another condition requires a user to have at least the balance of incoming HTLCs that cannot be persisted because the user was dishonest and the value of LedgerTime is larger than or equal to the HTLC's timelock + G. This condition might lead to steps becoming impossible but there are no steps of CommitHTLCsOnChain that can become possible.

$$\begin{aligned} \langle 2 \rangle 6. \quad \forall s \in \Sigma & : \quad \forall n \in ETP^{LedgerTime}(s) & : \\ & a(n) = FulfillHTLCsOnChain \Rightarrow \\ & n \in relETP^{LedgerTime}(s) \end{aligned}$$

PROOF: The action FulfillHTLCsOnChain can fulfill an HTLC on-chain as long as the HTLC's timelock + G is greater than LedgerTime. Because the action chooses a subset of fulfillable HTLCs as the HTLCs to fulfill, if the value of LedgerTime reaches an HTLC's timelock + G no new step becomes possible but only all steps that include fulfilling this HTLC become impossible.

If the preimage for an HTLC is learned when the value of LedgerTime is greater than the HTLC's timelock + G, the preimage is added to the set $uLatePreimages$. Thus, for each HTLC that can be fulfilled, the set $relETP^{LedgerTime}(s)$ must include the HTLC's timelock + G + 1. This is fulfilled by the definition of TimelockRegions of the module IdealChannel that is included by $relETP^{LedgerTime}(s)$.

$$\begin{aligned} \langle 2 \rangle 7. \quad \forall s \in \Sigma & : \quad \forall n \in ETP^{LedgerTime}(s) & : \\ & a(n) = ClosePaymentChannel \Rightarrow \\ & n \in relETP^{LedgerTime}(s) \end{aligned}$$

PROOF: The action ClosePaymentChannel contains a condition in the function ValidMapping that verifies that an HTLC can only be timed out if the value of

LedgerTime is at least the HTLCs timelock. Thus, steps in which the HTLC is timed out are only valid from a state on in which the value of LedgerTime is at least the HTLC's timelock. Therefore, the $relETP^{LedgerTime}(s)$ must include the HTLC's timelock. This is fulfilled by the definition of TimelockRegions of the module IdealChannel that is included by $relETP^{LedgerTime}(s)$. The function ValidMapping contains a check that an HTLC can only be persisted as long as the value of LedgerTime is lower than the HTLC's timelock + G. An increasing value of LedgerTime does not enable new steps to become possible.

If the preimage for an HTLC is learned when the value of LedgerTime is greater than the HTLC's timelock + G, the preimage is added to the set $uLatePreimages$. Thus, for each HTLC that can be fulfilled, the set $relETP^{LedgerTime}(s)$ must include the HTLC's timelock + G + 1. This is fulfilled by the definition of TimelockRegions of the module IdealChannel that is included by $relETP^{LedgerTime}(s)$.

$\langle 2 \rangle 8$. Q.E.D.

By $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, $\langle 2 \rangle 3$, $\langle 2 \rangle 4$, $\langle 2 \rangle 5$, $\langle 2 \rangle 6$, and $\langle 2 \rangle 7$ we have proven for all subactions of $NextI$ that $\forall x \in \mathcal{X}, s \in \Sigma : ETP^x(s) \subseteq relETP^x(s)$.

$\langle 1 \rangle 3$. Q.E.D.

By $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$ both statements of Assumption 1 are proven.