BitML^x: Secure Cross-chain Smart Contracts For Bitcoin-style Cryptocurrencies

Federico Badaloni *MPI-SP* federico.badaloni@mpi-sp.org Sebastian Holler

MPI-SP
sebastian.holler@mpi-sp.org

Chrysoula Oikonomou

Aristotle University of Thessaloniki
linen@protonmail.com

Pedro Moreno-Sanchez

IMDEA Software Institute

VISA Research

MPI-SP

pedro.moreno@imdea.org

Clara Schneidewind MPI-SP clara.schneidewind@mpi-sp.org

Abstract—A smart contract is an interactive program that governs funds in the realm of a single cryptocurrency. Yet, the many existing cryptocurrencies have spurred the design of crosschain applications that require interactions with multiple cryptocurrencies simultaneously. Currently, cross-chain applications are implemented as use-case-specific cryptographic protocols that serve as overlay to synchronize smart contract executions in the different cryptocurrencies. Hence, their design requires substantial expertise, as well as a security analysis in complex cryptographic frameworks.

In this work, we present $BitML^x$, the first domain-specific language for cross-chain smart contracts, enabling interactions with several users that hold funds across multiple Bitcoin-like cryptocurrencies. We contribute a compiler to automatically translate a $BitML^x$ contract into one contract per involved cryptocurrency and a user strategy that synchronizes the execution of these contracts. We prove that an honest user, who follows the prescribed strategy when interacting with the several contracts, ends up with at least as many funds as in the corresponding execution of the $BitML^x$ contract. Last, but not least, we implement the $BitML^x$ compiler and demonstrate its utility in the design of illustrative examples of cross-chain applications such as multi-chain donations or loans across different cryptocurrencies.

I. INTRODUCTION

A smart contract is an interactive program that controls cryptocurrency funds. Smart contracts are registered on a blockchain, a jointly maintained, append-only data structure that captures the system state of the cryptocurrency. The smart contract logic is then enforced as part of the consensus protocol that is executed by the cryptocurrency users to advance the system state by appending transactions (that represent user actions) to the blockchain.

While cryptocurrencies like Ethereum genuinely support complex stateful smart contracts that can be written in expressive (high-level) languages, the native smart contract support of other cryptocurrencies, foremost Bitcoin, is limited to basic payment conditions (e.g., digital signature-based authorization) expressed in a simple scripting language. This simple scripting functionality still gives rise to many interesting applications (e.g., escrows and lotteries) when composing multiple

of the abovementioned payment conditions and integrating them with a cryptographic protocol.

To facilitate the development of such applications, Bartoletti and Zunino contribute *BitML* [1], a high-level specification format for smart contracts in Bitcoin-style cryptocurrencies. *BitML* allows for defining smart contracts in terms of a simple process calculus that describes how the contract state evolves when interacting with a set of predefined contract users. For execution on a Bitcoin-style blockchain, the contract users can then compile a *BitML* smart contract into a cryptographic protocol over Bitcoin transactions that will realize the behavior of the high-level contract execution.

Problem. The execution of a smart contract is confined to the realm of a single blockchain and its underlying consensus protocol. This applies both to natively-supported and BitML smart contracts. However, there exist many different blockchains, which cater different use cases and many practical applications demand to govern funds at several blockchains simultaneously. As an illustrative example, a two-party swap [2] involves two users Alice and Bob who want to exchange funds that they own at different blockchains. This exchange must be atomic, meaning that Alice transfers their funds to Bob in the first blockchain if and only if Bob also transfers their funds to Alice in the second blockchain. Otherwise, both users should get their original funds refunded. More complex applications involve a larger number of users, each of which hold funds at potentially many blockchains (e.g., n-party swaps [3] is a generalization of the two-party swap where n parties swap funds held at k blockchains among each other).

State-of-the-art and challenges. Existing works in the literature tackle the abovementioned problem of cross-chain smart contracts [4] by contributing cryptographic protocols used as a synchronization overlay among smart contracts executed otherwise independently at different blockchains. These cryptographic protocols are application-specific meaning that they need to be adapted for each use case. The security of each of these protocols, hence, is proven from scratch,

requiring involved proofs in complex cryptographic proof frameworks [5], a task that is cumbersome and error-prone as demonstrated by the security flaws found so far in protocols proposed by both academia and industry [6]–[9].

Goal. In this work we want to answer the following question: Can we compile secure-by-design protocols over Bitcoin-style blockchains from a domain-specific language for cross-chain smart contracts?

Our approach. To achieve this goal, we define $BitML^x$, the first domain-specific language for cross-chain smart contracts. A smart contract expressed in BitMLx is compiled into (i) several BitML smart contracts, one per blockchain; and (ii) a strategy per honest user that defines how they must interact with the different BitML contracts to synchronize their execution. The smart contracts resulting from our compilation can be executed in Bitcoin-style blockchains. Therefore, our approach imposes minimal requirements to the underlying blockchains in that it only requires the support for basic scripting capabilities, and, hence, is compatible with a wide range of different blockchains. Building upon BitML, BitML^x smart contracts can be easily executed on UTXO-based cryptocurrencies with support for basic payment conditions that match the limited expressiveness of Bitcoin's scripting language (e.g., Dogecoin, Bitcoin Cash, Litecoin, or Cardano). Account-based cryptocurrencies with a (quasi) Turing-complete scripting language (such as Ethereum) could support BitML^x contracts by emulating the execution of BitML contracts.

Using *BitML*^x, cross-chain applications (e.g., atomic swaps) can be designed as a cross-chain smart contract abstracting away the cryptographic details to be handled when realizing applications with custom cross-currency protocols. In our approach, every honest user following the prescribed strategy is guaranteed that the execution of the several compiled *BitML* contracts ends up with them obtaining at least as many funds as in the corresponding execution of the *BitML*^x contract. While we find that *BitML*^x can express many interesting cross-chain applications, *BitML*^x is not a complete language for writing such applications. Its expressiveness is inherently limited by the underlying *BitML* language. Further, to enable secure cross-chain execution, *BitML*^x smart contracts, as opposed to *BitML* contracts, need to follow a slightly more restrictive structure. We discuss those limitations in detail in Section VII.

Our contributions. We make the following contributions:

- We introduce $BitML^x$, the first domain-specific language for cross-chain smart contracts (Section IV). $BitML^x$ permits expressing smart contracts among n users whose functionality requires interacting with k different blockchains.
- We contribute a compiler (Section V) to (i) translate $BitML^x$ contracts into k BitML contracts; and (ii) translate strategies to execute the $BitML^x$ contract into strategies to interact with the compiled BitML contracts. The crucial technical challenge lies in the safe replication of the smart contract steps in the k different blockchains. This must hold also in the presence of adversaries that can decide on the execution

order of scheduled actions and may even insert new actions that may conflict with the honest user's attempts to enforce synchronous contract executions.

- We prove the correctness of the $BitML^x$ compilation (Section VI). Our correctness result establishes that an honest user following the strategy prescribed by our compiler to execute the k BitML contracts will end up with at least as much funds as in the corresponding strategy in the $BitML^x$ contract.
- To showcase the practicality of our approach, we have implemented our compiler [10] and used it to implement and evaluate illustrative cross-chain applications (Section VIII) such as (i) multiblockchain donations, where a non-profit organization accepts donations in different denominations; (ii) multiblockchain payments with an exchange service, where a customer can decide to pay a service provider for its service in different denominations; and (iii) multi-blockchain loan with mediator, where a user borrows a loan in a certain denomination whereas holding funds in a different denomination as collateral. We discuss the scope and limitations of the BitML^x language for developing cross-currency applications (Section VII).

II. BACKGROUND

BitML in a nutshell. In [1], Bartoletti and Zunino introduce *BitML*, a domain-specific language for smart contracts in Bitcoin and other Bitcoin-style cryptocurrencies. As opposed to cryptocurrencies like Ethereum, Bitcoin does not allow for locking funds into interactive stateful smart contracts. Instead, Bitcoin transactions transfer the ownership of coins based on static spending conditions written in a simple scripting language. Integrating this basic mechanism with cryptographic protocols still allows for implementing a wide range of interesting applications on Bitcoin, such as lotteries or escrows.

BitML serves as a high-level specification format to express such applications while completely abstracting from Bitcoin transactions and cryptographic details. The BitML syntax permits defining contracts of the form $\{G\}C$ where G denotes the preconditions for the contract whereas C denotes the code encoding the contract logic according to which funds governed by the contract will be distributed. The preconditions G specify the funds that users need to provide to the contract $(A: v \otimes x)$ denotes the requirement of A to fund the contract with A bitcoins A and the secrets to which contracts users need to commit A: secret A denotes that A needs to commit to secret A.

A contract $C := D_1 + D_2 + \cdots + D_k$ is defined as a list of mutually exclusive execution choices represented by subcontracts D_i . Each sub-contract D_i has one of the following forms: (i) \vec{A} : D, denoting that \vec{A} must authorize the execution of D; (ii) reveal s then C, denoting that secret s must be revealed to continue with the execution of contract C; (iii) after t:D, denoting time t must be reached before executing \vec{D} ; (iv) split $\vec{v} | \vec{b} \rightarrow \vec{C}$, denoting that the contract funds are split according to the vector $\vec{v} | \vec{b} |$ and henceforth will be governed by the contracts as given by the vector \vec{C} ; and (v)

withdraw A, denoting that A can obtain the funds controlled by the contract.

For example, consider the following BitML contract that implements a so-called timed commitment where a user A commits to paying funds to another user B if B reveals a pre-agreed secret s:

```
\{A: !1 \ @x \mid B: secret s\}TC
TC = Exchange + Refund
Exchange = reveal s. withdraw B
Refund = after t. withdraw A
```

The contract preconditions determine that A needs to fund the contract with 1 $\overset{\circ}{B}$ (here x serves as the identifier of the concrete fund) and that B needs to commit to a secret named s. Once this happened (in which case we say that the contract was successfully stipulated), the funds can be withdrawn by B if s is revealed (i.e., Exchange clause). Alternatively, starting from time t, A can withdraw the funds (i.e., Refund clause). Such Refund clause is usually added to ensure that the funds of A cannot be locked indefinitely if B fails to collaborate.

In [1], Bartoletti and Zunino define a computational model of Bitcoin execution, as well as a symbolic model formally describing the execution of *BitML* contracts. They contribute a compiler that translates *BitML* contracts into a set of standard Bitcoin transactions and each symbolic honest user strategy into a computational strategy leveraging these transactions. Based on that, they provide a computational soundness result, showing that every protocol run in the computational model can be faithfully captured by a run in the symbolic model. This ensures that *BitML* contract guarantees proven in the symbolic model carry over the executions of the compiled protocol in the computational world.

III. KEY IDEAS

Natively supported smart contracts can only control funds from a single blockchain because their execution is enforced by the underlying blockchain-based consensus protocol. As a result, smart contracts on different blockchains (and therefore subject to different consensus protocols) execute independently. In practice, however, users are confronted with a heterogeneous environment of blockchains and wish to manage their funds in these systems in a unified and synchronized manner. Still, there exists no systematic way for developing cross-chain smart contracts that simultaneously govern funds on different blockchains.

In this section, we overview our approach to $BitML^x$, the first domain-specific language for cross-chain smart contracts. As a running example, we use a two-party atomic swap, a cross-chain application that has been largely studied in the literature and is widely used in practice [4]. In a two-party atomic swap, Alice and Bob want to exchange funds that they own on different blockchains (e.g., Alice owns 1 in Bitcoin whereas Bob owns 1 in Dogecoin). The outcome of the two-party atomic swap must be that Alice sends their bitcoins to Bob if and only if Bob sends Alice their dogecoins.

Using *BitML* to express two-party atomic swaps. A naive approach towards realizing a two-party atomic swap would be to make Alice and Bob lock their coins into timed commitment contracts on the chains where they own their funds and to condition the release of funds in both contracts to Bob revealing a secret s:

```
\{A: !1^{\Bar{B}} @x_A \mid B: secret s\} TC^{\Bar{B}}
TC^{\Bar{B}} = Exchange^{\Bar{B}} + Refund^{\Bar{B}}
Exchange^{\Bar{B}} = reveal s . withdraw B
Refund^{\Bar{B}} = after t . withdraw A
\{B: !1^{\Bar{D}} @x_B \mid B: secret s\} TC^{\Bar{D}}
TC^{\Bar{D}} = Exchange^{\Bar{D}} + Refund^{\Bar{D}}
Exchange^{\Bar{D}} = reveal s . withdraw A
Refund^{\Bar{D}} = after t . withdraw B
```

If both users are honest, the execution of these two contracts does actually lead to a successful swap: Imagine that Alice and Bob agree to swap the coins. In such case, Bob can reveal the secret s to execute the subcontract $Exchange^{\vec{B}}$, letting Alice learn s and consequently execute the subcontract $Exchange^{\vec{D}}$. Similarly, this approach allows honest users to safely abort the swap: Imagine that Bob does not wish to swap the coins. In such case, Bob does not reveal the secret s, and after time t expires, Alice (correspondingly Bob) can get refunded by executing the contract $Refund^{\vec{D}}$ (correspondingly $Refund^{\vec{D}}$).

Issues with malicious users. The challenges appear when any of two users does not obey the prescribed strategy to execute the *BitML* contracts. Note that the contracts TC^{\square} and TC^{\square} reside in two blockchains and therefore execute independently. The coordination of actions in different contracts relies on the users observing and replicating smart contract interactions. However, the blockchain-based consensus does not easily allow for atomic replication of actions observed on another blockchain: Blockchain (and so, in particular smart contract) interactions conducted by (honest) users do not immediately come into effect but get scheduled for execution. A potentially malicious scheduler (e.g., a Bitcoin miner) then decides on the execution order of scheduled actions and may even insert new actions. Attempts to replicate an action observed on another blockchain, hence, may be preempted by the execution of other, conflicting actions.

For instance, imagine that a malicious user Bob collaborating with the scheduler waits until time t and only then reveals

s and executes $Exchange^{\Bar{D}}$. Even if Alice observes this action and tries to replicate it by scheduling $Exchange^{\Bar{D}}$ for execution, it is not guaranteed that $Exchange^{\Bar{D}}$ will get successfully executed. Instead, Bob may schedule $Refund^{\Bar{D}}$ and the malicious scheduler may execute it before $Exchange^{\Bar{D}}$, hence making Bob receive the funds of both contracts.

A. Our Approach

In this work, we develop a general solution to the described atomic replication problem. To this end, we design $BitML^x$, a novel domain-specific language for expressing cross-currency smart contracts and show how to compile $BitML^x$ contracts into independently executing BitML contracts, which support safe replication by design.

Using $BitML^x$, we can express the two-party atomic swap as follows:

```
\begin{array}{ll} Swap^x &= \{A: 1\cline{10}B: 1\cline{10}\}Pay^x \Leftrightarrow Abort^x \\ Pay^x &= \cline{10}w(1\cline{10} \to A, \ 1\cline{10} \to B) \\ Abort^x &= \cline{10}w(1\cline{10} \to A, \ 1\cline{10} \to B) \end{array}
```

Departing from BitML and crucial to securely support crosschain applications (as we explain later), a contract in $BitML^x$ is defined as a priority choice between two clauses. In the running example, every contract user has the opportunity to execute the move Pay^x (i.e., high priority move), and only if all users decide to skip this option, the alternative clause $Abort^x$ (i.e., low priority move) will be available. The code Pay^x simultaneously enables A to withdraw 1^n_V and B to withdraw 1^n_V . The clause $Abort^x$ enables A and B to reclaim their originally deposited funds. Effectively, the contract implements an atomic swap between users A (holding 1^n_V): Either both funds are transferred to the respective counterparty or both funds are refunded.

Compilation into two *BitML* contracts. We compile the $BitML^x$ contract into two BitML contracts, each of which can be executed on one of the involved blockchains, in our running example Bitcoin and Dogecoin. Moreover, we compile honest $BitML^x$ user strategies into a strategies describing what steps to perform at each of the BitML contracts. Next, we overview the challenges and design rationale of our compilation.

Intuitively, one can think of a priority choice in $BitML^x$ as a temporal priority between the two moves in the sense that there is a time window where only the high priority move can be taken, and after which only the low priority move is viable to the detriment of the high priority move. To realize such a behavior, our compilation first offers the high priority move on both blockchains. To execute such a high priority move, an honest user is always required to move the contracts at both blockchains simultaneously. For instance, in Fig. 1, any of the users can take both, the choice of withdraw Bin Bitcoin and the choice of withdraw A in Dogecoin. If one of the users does both moves simultaneously, then the contract $Swap^x$ is faithfully executed in both blockchains and the execution terminates. Otherwise, we require a mechanism to handle the situation where the high priority move is done only at one of the chains, as we describe next.

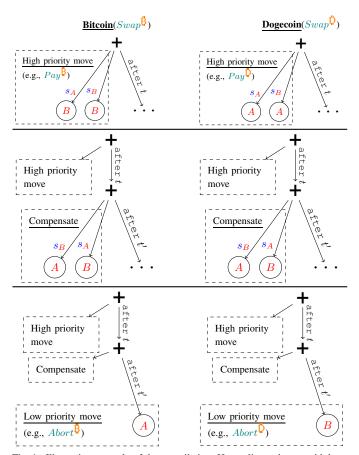


Fig. 1. Illustrative example of the compilation. Here, elipses denote withdraw funds by the corresponding participant; dashed boxes represent logical steps in our compilation. The underlined name indicates the active phase. We assume t' > t so that the honest user has enough time to execute *Compensate*.

Instead of enabling an honest user to replicate partial moves of a malicious user to ensure synchronous execution of the two BitML contracts, we design a compilation that allows honest users to get compensated when observing nonsynchronous moves. For this, it is crucial that users can be held accountable for the moves that they make. We ensure this in our compilation by preparing individualized move options for each user that require the user to reveal a personalized secret (called step secret) when taking a move. A step secret, hence identifies both the user and the move they took. For instance, in our running example, A must reveal the step-secret s_A to move $Swap^{\square}$ into withdraw A. Similarly, B must reveal the step-secret s_B to do these moves instead.

With the step-secrets in place, after the high priority move times out, the contract can be moved into a compensation phase where an honest user can claim misbehavior of other users. Such claim is done by providing the step secret revealed by the misbehaving user that proves that they made the high priority move on the other chain, while they did not on the current chain (since otherwise they would not have entered the compensation phase there). For instance, in Fig. 1 assume now that B makes the move of withdraw B in $Swap^{\frac{B}{b}}$ but

does not make the move of withdraw A in $Swap^{\square}$ (i.e., a malicious B wants to obtain 1 without paying 1 to A). In such case, after the high priority choice in the contract $Swap^{\square}$ times out, the honest participant A can get compensated by revealing the secret s_B learned from the other chain.

After the compensation phase, the contract execution moves to a phase where only the lower priority choice is available. In our running example, assume now that none of the users decided to synchronously move the contract in both chains to the high priority move, nor needed to get compensated after a partial move from the other party. Then, after enough time to execute the compensation has passed, the honest user can synchronously move both contracts to the low priority choice.

Note that, to ensure a safe execution, an honest user needs to obey certain rules, such as only performing synchronous moves themselves, and always perform compensations when observing non-synchronous moves. To see this, as before consider a malicious Bob that makes the move of withdraw B in $Swap^{\square}$ but does not make the move of withdraw A in $Swap^{\square}$. The strategy of A must be such that it executes the compensation as soon as it gets available. Otherwise, after enough time to execute compensation has passed, Bob successfully gets away with 1^{\square} without paying 1^{\square} to A.

B. Discussion

Generalization to other contracts. To ease the explanation, the running example is limited to a contract for a single priority choice. However, the described procedure can be used to compile more complex smart contracts that consists of multiple execution steps (c.f. Section VIII). This is possible because, after executing each of the priority choices, the execution of the contract enters into a safe state from the point of view of the honest user. To see this, observe that in the simplest case we have that the priority choice was synchronously executed, either with the high priority move or the low priority one. For the case where the priority choice was asynchronously executed, assume w.l.o.g. that the high priority choice was taken on chain \mathbb{B}_1 but not taken on chain \mathbb{B}_2 (the other case is symmetric). In such case, the honest user can enter the compensation phase in \mathbb{B}_2 and obtain the complete contract balance. Therefore, the honest user obtained the best possible outcome on chain \mathbb{B}_2 . In chain \mathbb{B}_1 , the original contract keeps being executed, according to the original contract's logic. This will result in an honest user obtaining a valid outcome (according to the original contract) on \mathbb{B}_1 and the maximal outcome on \mathbb{B}_2 , which leaves them at least as good as in a synchronous execution in both chains.

Collateral. In the 2-party case, the compensation ensures that the non-cheating party has a beneficial outcome because they obtain all funds locked in the compensated blockchains in the case of cheating. To generalize this idea to n-party contracts, users will initially lock additional collateral that can be used for compensating honest users in case of cheating and released back after honest execution. Assuming that the $BitML^x$ contract balance is b_i for the i-th chain, each user

```
G := |A :!B @ \vec{x} | \text{deposit of } B, \text{ expected from } A \\ |A : \text{secret } s | \text{committed secret by } A \\ |G||G' | \text{composition} \\ |B = [v_1\mathbb{B}_1, \dots, v_k\mathbb{B}_k] | \text{Balance}
```

Fig. 2. Syntax of preconditions of $BitML^x$ contracts.

needs to additionally lock $(n-2) \cdot b_i$ coins for the *i*-th chain as collateral. This suffices to ensure that every honest user can always be compensated in case of misbehavior at each of the chains: the misbehaving user's collateral can be used to compensate n-2 users. The amount already held by the contract can be used to compensate the remaining honest user.

Generalization to k chains. Our approach seamlessly generalizes to smart contracts that control funds in k>2 different chains by allowing honest users to compensate whenever entering a compensation phase in a chain \mathbb{B}_i while obtaining the step secret of another user for the respective step. The step secret indicates that at least one chain moved to the high priority choice while \mathbb{B}_i (having entered the compensation phase) did not. Compensating on all chains \mathbb{B}_i that entered the compensation phase, safely aborts the execution on these chains (leaving honest users with the most beneficial outcome for them). All remaining chains are ensured to have taken the high priority choice synchronously and can resume execution.

Correctness. We formally define the semantics of $BitML^x$ and specify (i) a contract compiler that translates a $BitML^x$ contract into k BitML contracts (running on k different Bitcoin-style blockchains) and (ii) a strategy compiler that transforms a strategy of an honest user interacting with a $BitML^x$ contract into a strategy for that user to interact with the k BitML contracts resulting from the contract compilation. We provide a generic correctness result for the compilation that shows that an honest user, when executing the compiled BitML strategy, will always end up in a state that allows them to cash out at least as many funds as they could have obtained from execution of the corresponding $BitML^x$ strategy (executing the $BitML^x$ contract).

IV. The $BitML^x$ Language

In the following, we assume a set Part of *participants*, ranged over by A, B, Further, we denote by Hon \subseteq Part a non-empty set of *honest* participants. We will use the following naming conventions: (i) *chains* denoted by \mathbb{B} ; (ii) *deposits* denoted by x. We denote by \vec{x} a finite sequence of deposit names, and similar notation is adopted for sequences of other kinds; (iii) *secrets* denoted by s.

A. Syntax

Contract preconditions. The syntax of contract preconditions is shown in Fig. 2. Similar to BitML, A: secret s requires A to commit to a (randomly chosen) secret with name s. During the execution of the contract C, A can

```
\begin{array}{lll} C ::= D \Leftrightarrow C & \text{top-level contracts} \\ | \ \text{withdraw} \ \vec{\mathbf{B}} \to \vec{A} & \text{transfer the balances} \ \vec{\mathbf{B}} \ \text{to} \ \vec{A} \\ D ::= \ \text{withdraw} \ \vec{\mathbf{B}} \to \vec{A} & \text{guarded contracts} \\ | \ \text{split} \ \vec{\mathbf{B}} \to \vec{C} & \text{split the balance} \ (|\vec{\mathbf{B}}| = |\vec{C}|) \\ | \ \vec{A} : \ D & \text{wait for authorization from} \ \vec{A} \\ | \ \text{reveal} \ \vec{s} \ \text{if} \ p \ \text{then} \ C & \text{collect secrets} \ \vec{s} \end{array}
```

Fig. 3. Syntax of $BitML^x$ contracts.

```
E ::=
                                                  contract expression
p := true
                 conjunction
                                          N
                                                  32-bit constant
    \mid p \wedge p
    |\neg p|
                 negation
                                     |s|
                                                  length of a secret
    \mid E = E
                                     \mid E + E
                 equality
                                                  addition
                                     \mid E - E
    \mid E < E
                 less than
                                                 subtraction
```

Fig. 4. Syntax of $BitML^x$ reveal conditions (as defined in [1]).

decide whether to reveal s to other participants. Additionally, the preconditions of a $BitML^x$ contract specify the deposits required to stipulate the contract and the secrets to which contract users should commit prior to execution. As opposed to BitML, in $BitML^x$ the contract deposits of users stem from multiple blockchains, and hence are provided as a vector: The precondition $A:!B@\vec{x}$ requires A to own $v_i=B[B_i]$ coins in a deposit x_i at chain B_i . Finally, $BitML^x$ contracts feature a single precondition of type (t_0, κ_0) that specifies the contract execution parameters t_0 , denoting the contract's starting time and κ_0 , denoting a unique contract identifier. Intuitively, $BitML^x$ contracts require a starting time to safely synchronize the cross-blockchain stipulation of the different BitML contracts.

Contracts. The syntax for $BitML^x$ contracts is given in Fig. 3. A contract C is given either by a contract withdraw $\vec{B} \to \vec{A}$ or a *priority choice* between a guarded contract D and the continuation contract C'. A contract withdraw $\vec{B} \to \vec{A}$ represents the default case of a priority choice that distributes the contract funds to (possibly only part of) the contract users. It denotes that each user $A_i \in \vec{A}$ obtains a balance from the involved blockchains as defined by $B_i = [v_{i,1}\mathbb{B}_1, \dots, v_{i,k}\mathbb{B}_k]$.

The left (high priority) branch of a priority choice must be a guarded contract D. A guarded contract can be of one of the following forms: (i) withdraw $\vec{B} \to \vec{A}$ (same as the default case) distributes the contract funds on the different chains to the contract users (ii) split $\vec{B} \to \vec{C}$ splits the contract balance according to the vector $\vec{B} = (B_1, \ldots, B_n)$ and continues executing contracts $C_1, \ldots C_n$ as given by \vec{C} with the respective balances B_1, \ldots, B_n . (iii) $\vec{A} : D$ waits for the authorization of users \vec{A} and (iv) reveal \vec{s} if p then C given the secrets in \vec{s} being revealed, evaluates the condition p (potentially referencing those secrets) and continues to execute as C if p evaluates to true. The syntax of conditions is shown in Fig. 4.

$$\Gamma ::= \langle C, \mathbf{B} \rangle_{\kappa} \mid \langle A, \mathbf{B} \rangle_{\kappa} \mid \{A : s \# N\} \mid A : s \# N \mid A : [\kappa \triangleright D]$$

Fig. 5. BitML^x configurations for contact execution

B. Semantics

A configuration Γ (cf. Fig. 5) records the state of currently executing $BitML^x$ contracts as well as contract-relevant (user) interactions. We define the semantics of $BitML^x$ as a relation $\Gamma \xrightarrow{\alpha} \Gamma'$ that describes how a configuration Γ changes when executing (user) action α .

To initiate the execution of a $BitML^x$ contract, users can advertise a contract, commit to the secrets prescribed in G and authorize its stipulation. A stipulation results in an active contract $\langle C, B \rangle_{\kappa}$, which evolves according to the semantic rules provided in Fig. 6. Contracts of the form $D \Rightarrow C$ either execute according to the high-priority choice D (rules D-Withdraw, D-Split, D-Reveal, and D-Auth) or discard the high-priority choice executing the *skip* action (rule *C-Skip*). The D-Withdraw rule splits the balance B of a contract $\langle \text{withdraw } \overrightarrow{B} \to \overrightarrow{A} \Leftrightarrow C, \overrightarrow{B} \rangle_{\kappa}$ according to the vector \overrightarrow{B} and assigns these balances to the users in \vec{A} resulting in terminal contracts of the form $\langle A_i, B_i \rangle_{\kappa_i}$ that assign the balances to the corresponding users A_i . The *D-Split* rule proceeds similarly but transfers the balances to subcontracts $\langle C_i, B_i \rangle_{\kappa_i}$ as specified by a vector \vec{C} . The *D-Reveal* rule allows for evolving contract (reveal \vec{s} if p then $C' \Rightarrow C, \mathbf{B}\rangle_{\kappa}$ into a contract $\langle C', \mathbf{B} \rangle_{\kappa'}$ provided that the secrets in \vec{s} have been revealed (indicated by elements A_i : $s_i \# N_i$ in the configuration) and that the condition p evaluates to true ($[p]_{\Delta} = true$). Users can reveal secrets that they committed to previously any time (cf. rule *RevealSecret*). Revealing a secret, in particular, reveals the secret's length N_i which can be referenced by the condition p (cg. Fig. 4). Correspondingly, the condition p is evaluated in an environment that resolves this reference for revealed secrets. Finally, rule *D-Auth* allows for evolving contracts $\langle \vec{A} : D \Rightarrow C, \vec{B} \rangle_{\kappa}$ whose high priority branch requires authorization from the users in \vec{A} . Such authorizations are indicated by configuration elements of the form $A_i : [\kappa \triangleright D]$, which can be added by the corresponding user (cf. rule A-Signature). If such authorizations are provided by all users, $\langle \vec{\mathbf{A}} : D \Rightarrow C, \mathbf{B} \rangle_{\kappa}$ evolves the same as $\langle D \Rightarrow C, \mathbf{B} \rangle_{\kappa}$.

The only contract that can occur standalone (outside a priority choice) is of the form $\langle \text{withdraw } \vec{B} \to \vec{A}, B \rangle_{\kappa}$ and evolves exactly as it would do when appearing as the high-priority branch of a priority choice (cf. rule *C-Withdraw*).

Strategies and execution model. The $BitML^x$ semantics specifies possible execution steps of $BitML^x$ smart contracts. However, many of these execution steps are restricted to be performed by specific users A (as indicated by labels of the form $A:\alpha$). To characterize valid $BitML^x$ smart contract interactions, we hence consider the $BitML^x$ semantics in conjunction with the strategies of the different contract users (as it is also done for BitML).

$$\frac{D = \operatorname{withdraw} \stackrel{\circ}{\mathbb{B}} \to \stackrel{\circ}{A} \qquad \stackrel{\circ}{\mathbb{B}} = \operatorname{B}_1, \ldots, \operatorname{B}_n \qquad \stackrel{\circ}{A} = A_1, \ldots, A_n}{\langle D \Leftrightarrow C, \mathsf{B} \rangle_{\kappa} \mid \Gamma} \stackrel{\operatorname{dwithdraw}(\kappa)}{\longrightarrow} \prod_{i=1}^n \langle A_i, \mathsf{B}_i \rangle_{\kappa_i} \mid \Gamma} D - \operatorname{Withdraw}$$

$$\frac{D = \operatorname{split} \stackrel{\circ}{\mathbb{B}} \to \stackrel{\circ}{C} \qquad \stackrel{\circ}{\mathbb{B}} = \operatorname{B}_1, \ldots, \operatorname{B}_n \qquad \stackrel{\circ}{C} = C_1, \ldots, C_k \qquad \kappa_1, \ldots, \kappa_k \text{ fresh}}{\langle D \Leftrightarrow C, \mathsf{B} \rangle_{\kappa} \mid \Gamma} D - \operatorname{Split}$$

$$\frac{D = \operatorname{reveal} \stackrel{\circ}{\mathcal{S}} \operatorname{if} p \operatorname{then} C' \qquad \stackrel{\circ}{\mathcal{S}} = s_1, \ldots, s_k \qquad \Delta = \prod_{i=1}^k A : s_i \# N_i \qquad \llbracket p \rrbracket_{\Delta} = \operatorname{true} \qquad \kappa' \text{ fresh}}{\langle D \Leftrightarrow C, \mathsf{B} \rangle_{\kappa} \mid \Delta \mid \Gamma} D - \operatorname{Reveal}$$

$$\frac{\langle D \Leftrightarrow C, \mathsf{B} \rangle_{\kappa} \mid \Delta \mid \Gamma \xrightarrow{\operatorname{reveal}(\kappa)} \langle C', \mathsf{B} \rangle_{\kappa'} \mid \Gamma}{\langle A_i \in \stackrel{\circ}{\mathcal{A}} \qquad \langle A_i \in \stackrel{\circ}{\mathcal{A}} \qquad \langle A_i \in \stackrel{\circ}{\mathcal{A}} \qquad \rangle}{\langle A_i \in \stackrel{\circ}{\mathcal{A}} \qquad \langle A_i \in \stackrel{\circ}{\mathcal{A}} \qquad \rangle} D - \operatorname{Auth}}$$

$$\frac{\langle C = \operatorname{withdraw} \stackrel{\circ}{\mathcal{B}} \to \stackrel{\circ}{\mathcal{A}} \qquad \stackrel{\circ}{\mathcal{B}} = \operatorname{B}_1, \ldots, \operatorname{B}_n \qquad \stackrel{\circ}{\mathcal{A}} = A_1, \ldots, A_n}{\langle A_i \in \mathcal{A} \mid \Gamma \stackrel{\circ}{\mathcal{A}} \bowtie \Gamma' \qquad \rangle} C - \operatorname{Withdraw}$$

$$\frac{\langle C, \mathsf{B} \rangle_{\kappa} \mid \Gamma \xrightarrow{\operatorname{cwithdraw}(\kappa)} \prod_{i=1}^n \langle A_i, \mathsf{B}_i \rangle_{\kappa_i} \mid \Gamma}{\langle A_i, \mathsf{B}_i \rangle_{\kappa_i} \mid \Gamma} C - \operatorname{Skip}$$

$$\frac{\langle A \in \stackrel{\circ}{\mathcal{A}} \qquad \rangle}{\langle A \in \stackrel{\circ}{\mathcal{A}} \qquad \rangle} \stackrel{\circ}{\langle A \in \stackrel{\circ}{\mathcal{A}} \qquad \rangle} \langle A \cap \mathsf{Signature}$$

$$\frac{\langle A \in \stackrel{\circ}{\mathcal{A}} \qquad \rangle}{\langle A \cap \mathcal{A} \mapsto \langle A \cap \mathcal{A} \cap$$

Fig. 6. BitML^x semantics for active contracts

Formally, we represent a $BitML^x$ strategy of user A as a function Σ_A^x that takes as input a run R^x of the form $\Gamma_0 \xrightarrow{\alpha_0} \dots \xrightarrow{\alpha_{n-1}} \Gamma_n$ and outputs a set of actions $A = \Sigma_A^x(R^x)$. For Σ_A^x to be valid it needs to hold that (i) if $\alpha \in \Sigma_A^x(R^x)$ then there exists Γ such that $\Gamma_n \xrightarrow{\alpha} \Gamma$ (so α is a valid move w.r.t the semantics) (ii) if $B: \alpha \in \Sigma_A^x(R^x)$ then B = A (so Σ_A^x only outputs actions that A is entitled to perform) (iii) if $\alpha \in \Sigma_A^x(R^x)$ and $\hat{R}^x = R^x \xrightarrow{\hat{\alpha}} \hat{\Gamma}$ is a valid extension of R^x then also $\alpha \in \Sigma_A^x(\hat{R}^x)$.

The last restriction captures that Σ_A^x is *persistent*, meaning that users cannot take back actions that they wanted to execute at some point. This requirement becomes relevant when considering that actions output by Σ_A^x are not executed instantaneously but are handed to a malicious scheduler (e.g., a miner) who decides on the execution order.

More precisely, we assume the existence of an adversarial strategy Σ^x_{Adv} that models the behavior of malicious protocol participants and the malicious scheduler. Such a strategy Σ^x_{Adv} , in addition to the run R^x , gets as input the output $A = \Sigma^x_A(R^x)$ of the honest user strategy and based on that outputs the next action α to append to R^x . Σ^x_{Adv} , similar to honest user strategies, is limited to only output actions $\alpha = \Sigma^x_{\text{Adv}}(R^x, A)$ that are valid extensions of R^x according to the $BitML^x$ semantics. Further, Σ^x_{Adv} may not schedule any actions $A: \alpha$ unless those are included in $\Sigma^x_A(R^x)$.

The same holds for actions $\alpha = skip(\kappa)$, reflecting that the honest user needs to agree to skipping a high priority branch of a priority choice contract. We say that a run $R^x = \Gamma_0 \xrightarrow{\alpha_0} \Gamma_1 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{n-1}} \Gamma_n$ conforms to Σ_A^x (written $\Sigma_A^x \vdash R^x$) if there exists some adversarial strategy Σ_{Adv}^x such that R^x results from the interactions of Σ_A^x with Σ_{Adv}^x , meaning that $\alpha_i = \Sigma_{\mathsf{Adv}}^x(R_i^x, A_i)$ for $R_i^x = \Gamma_0 \xrightarrow{\alpha_0} \dots \xrightarrow{\alpha_{i-1}} \Gamma_i$ and $A_i = \Sigma_A^x(R_i^x)$ for all $i = 0, \dots, n-1$.

A full specification of the $BitML^x$ semantics can be found in Appendix B.

V. COMPILATON

In this section, we describe how to compile a $BitML^x$ contract C operating across k blockchains into k individual BitML contracts $C_1, \ldots C_k$ running on these chains. Further, we show how to translate an honest user strategy Σ_A^x for a participant A of C into a BitML strategy enforcing the synchronous execution of $C_1, \ldots C_k$ that corresponds to an execution of C conforming to Σ_A^x .

A. Compilation of contracts

To facilitate compilation, we define a compiler state Ω that carries relevant compilation information. The compiler state Ω gets instantiated based on the information from the contract advertisement $\{G\}C$ (e.g., with the contract's starting time

 t_0). During the compilation, Ω gets updated reflecting that certain compilation parameters depend on the position of a compiled subcontract within the original contract's syntax tree. This applies in particular to the timelocks used to implement the priority choices, which increase with each tree level.

Priority choices. Remember that a $BitML^x$ priority choice is of the form $D \Leftrightarrow C'$. The compiler translates such a contract into two clauses: (i) one clause corresponding to the high priority contract D and (ii) one clause implementing the compensation phase followed by the execution of low-priority contract C'.

To ensure that an honest user can not be hindered from taking clause (i), clause (ii) is guarded by a timelock t. The timelock makes clause (ii) only available starting from time t, giving the honest user sufficient time to claim clause (i) before. The time t is accessed from the compiler state Ω where it increases with each level that the compilation descends into the syntax tree, ensuring that the timelocks increase with increasing depth.

Following the same idea, to ensure a that an honest user has sufficient time to compensate a malicious user, clause (ii) consists of a choice between a compensation contract and a continuation contract guarded with timelock $t+\hat{\delta}$. The parameter $\hat{\delta}>0$ is a system parameter, intuitively denoting the maximal time that a malicious scheduler can delay the execution of an honest user action.

E.g., our running example $C := Pay^x \Rightarrow Refund^x$ is translated into

$$\begin{split} \mathcal{F}_{D}^{\mathbb{B}}(Pay^{x},\Omega) \; + \; & \text{after} \; t : (\\ \mathcal{F}_{Compen}^{\mathbb{B}}(\Omega) \; + \; & \text{after} \; (t+\hat{\delta}) : \mathcal{F}_{C}(Refund^{x},\Omega)) \end{split}$$

Here, $\mathcal{F}_D^{\mathbb{B}}$ denotes the compilation of a $BitML^x$ guarded contract into a BitML contract for chain \mathbb{B} ; $\mathcal{F}_{Compen}^{\mathbb{B}}$ denotes the compilation of the compensation mechanism into BitML for chain \mathbb{B} ; and \mathcal{F}_C denotes the compilation of the continuation contract into a BitML contract for chain \mathbb{B} .

Compiling $BitML^x$ guarded contracts. The compilation of a $BitML^x$ guarded contract D into a BitML contract depends on the concrete form of D. We illustrate the compilation using the example of a withdraw contract and refer to Appendix C for a detailed description of the other cases.

A $BitML^x$ withdraw contract is compiled into a BitML contract for target chain \mathbb{B} according to the following rule:

$$\begin{split} D &= \text{withdraw } \vec{\mathbf{B}} \to \vec{A} \\ S &= \{ s_{\kappa}^{A} \in \Omega.stepSecrets : \kappa = \Omega.currentLabel \} \\ \vec{\mu} &= \mathcal{F}_{Out}^{\mathbb{B}}(\vec{A}, \vec{\mathbf{B}}, \Omega) \\ \vec{C^{\mathbb{B}}} &= [\text{withdraw } A_{i} \mid A_{i} \leftarrow \Omega.participants] \\ \hline \mathcal{F}_{D}^{\mathbb{B}}(D, \Omega) &= \sum_{s_{\kappa}^{A} \in S} \text{reveal } s_{\kappa}^{A} \text{ . split } \vec{\mu} \to \vec{C^{\mathbb{B}}} \end{split}$$

On the high level, the compilation consists of a choice guarded by a step secret s_{κ}^{A} for each contract user A. To this end, at the beginning of the compilation, Ω got initialized with a set of step secrets ($\Omega.stepSecrets$) for each contract user A and position κ in the original's contract's syntax tree.

The current position κ of the compiled priority choice can be accessed from Ω (by $\Omega.currentLabel$) and is updated as the compiler traverses the syntax tree. The split distributes the balance of the contract as computed by $\mathcal{F}_{Out}^{\mathbb{B}}(\vec{A}, \vec{B}, \Omega)$, which namely (1) assigns the collateral back to each participant (accessed by $\Omega.participants$); and (2) additionally assigns part of the contract's balance to each user as encoded in \vec{B} of the $BitML^x$ withdraw statement.

As an illustrative example, the output of $\mathcal{F}_D^\mathbb{B}(Pay^x,\Omega)$ for the chain \mathbb{B} is as shown below. Here, s_κ^A is the step secret for A for choosing the high-priority choice where $\kappa = [\kappa_0, L]$, encoding the first left (L) choice of a contract with identifier κ_0 . Similarly, s_κ^B is the step secret for B and label κ .

$$\begin{aligned} \text{reveal } s_{\kappa}^{\pmb{A}}.\text{split}[1, \text{withdraw } \pmb{B}] \\ & + \\ \text{reveal } s_{\kappa}^{\pmb{B}}.\text{split}[1, \text{withdraw } \pmb{B}] \end{aligned}$$

The compilation of split, reveal, and authorization contracts follows the same pattern.

Compiling the compensation mechanism. The compensation mechanism is designed to compensate honest users $\vec{A_i}$ when a dishonest participant $A_j \notin \vec{A_i}$ did not do the high-priority move in all chains. In such case, the misbehaving user must have revealed their corresponding step secret $s_k^{A_j}$ to execute the move in some chains whereas in the other chains, honest users can enter the compensation phase at time $t + \hat{\delta}$. At this point, the compilation ensures the existence of the following compensation contract:

$$\sum_i$$
 reveal $s^{m{A_j}}_{\kappa}$. (split $ec{v_i}
ightarrow ec{C_i}$)

Each clause in the sum requires to reveal the step secret $s_{\kappa}^{A_j}$ corresponding to the misbehaving user A_j . Then, the total balance of the contract (i.e., $\vec{v_i}$) is equally split among the honest participants. This is encoded in $C_i := [\text{withdraw } \vec{A_i}]$. Remember that, at the stipulation of the contract, every user must not only include their deposits but also a collateral of $(n-2) \cdot b_0$ (where b_0 denotes the initial contract balance). Therefore, a contract, at any point of execution, is ensured to effectively hold balance n*(n-2)*b+b (where b is the contract's logical balance). This ensures, in particular, that when entering the compensation contract, (n-1) users (all users but the misbehaving one) can retrieve (n-2)*b+b corresponding to a refund of the provided collateral for that subcontract as well as the maximal payout b for the subcontract.

As an illustrative example, the output of $\mathcal{F}_{Compen}^{\mathbb{B}}(\Omega)$ in our running example for target chain \mathbb{B} is as shown below. A gets the contract balance if B's step secret was revealed in another chain. The other choice is symmetric.

reveal
$$s_{\kappa}^{\pmb{A}}. \text{split}[1, \text{withdraw } \pmb{B}] \\ + \\ \text{reveal } s_{\kappa}^{\pmb{B}}. \text{split}[1, \text{withdraw } \pmb{A}]$$

B. Compilation of strategies

To mimic the behavior of a $BitML^x$ strategy Σ_A^x , a corresponding low-level strategy Σ_A , operating on the compiled BitML contract on different chains, needs to fulfill two functions: (1) (synchronously) perform moves scheduled by Σ_A^x (2) maintain a synchronous execution state by detecting nonsynchronous moves of other users and safely aborting execution on chains that deviate from the synchronous execution.

For Σ_A to provide the first function, we need to map a BitML run R to its corresponding $BitML^x$ run R^x . To this end, we will introduce a coherence relation \sim_A , relating a BitML run R and a $BitML^x$ run R^x , such that intuitively, $R^x \sim_A R$ denotes that from the perspective of user A, R represents an execution corresponding to $BitML^x$ run R^x .

Intermediate semantics. As a first step towards defining coherence, we establish a direct link between $BitML^x$ and BitML executions by defining an intermediate semantics $\Gamma^{||} \stackrel{\alpha}{\to} \hat{\Gamma}^{||}$ that describes the concurrent execution of compiled contracts $C_1, \ldots C_k$ executing on chains $\mathbb{B}_1, \ldots \mathbb{B}_k$ in terms of the original $BitML^x$ contract C. The intermediate semantics has a direct correspondence to the BitML semantics, more precisely, there are functions $f_c()$ and $f_a()$ such that

$$\Gamma_0^{||} \xrightarrow{\alpha_0} \dots \xrightarrow{\alpha_{n-1}} \Gamma_n^{||} \Leftrightarrow f_c(\Gamma_0) \xrightarrow{f_a(\alpha_0)} \dots \xrightarrow{f_a(\alpha_{n-1})} f_c(\Gamma_n)$$

Elements in an intermediate configuration $\Gamma^{||}$ are of the form $\langle C, \pi \rangle_{\kappa,x}^{\mathbb{B}}$, indicating that they represent the execution of a BitML contract corresponding to the compilation of $BitML^x$ contract C (with $BitML^x$ identifier κ) in blockchain \mathbb{B} with state π . The identifier x establishes a direct link to the active BitML contract matching $\langle C, \pi \rangle_{\kappa,x}^{\mathbb{B}}$ on the BitML level. The state π indicates the contract's execution phase (e.g., whether C is idle, its compensation phase has been entered, or it got compensated). Due to the direct correspondence between the BitML and the intermediate semantics, we will, henceforth, state all results in terms of the intermediate semantics, using $R^{||}$ to denote runs in the intermediate semantics and $\Sigma_A^{||}$ to denote the honest user strategy based on the intermediate semantics. A full specification of the intermediate semantics can be found in Appendix D.

Coherence. Defining relation \sim_A is challenging because there exist many possible runs $R^{||}$ that represent (potentially incomplete) executions of a run R^x : Users may take the high-priority branches of priority choices in a some chain \mathbb{B}_i and in this way, run ahead with the execution of the *BitML* contract on \mathbb{B}_i without (immediately) matching these moves on the other blockchains.

In such cases, it is still ensured that, eventually, the moves taken by the contract in the most advanced chain \mathbb{B}_i can be matched because, either the execution on all other chains \mathbb{B}_j will advance to the same state, or the execution on \mathbb{B}_j will enter the compensation phase and hence the execution on \mathbb{B}_j can be safely aborted.

For this reason, \sim_A will associate a run $R^{||}$ with a $BitML^x$ run R^x reflecting the "most-advanced" executions of compiled

contracts $C_1, \ldots C_k$ in $R^{||}$. Since those contracts, during execution, can be split in independently executing subcontracts, it is not guaranteed that there is a single blockchain \mathbb{B}_i featuring the most advanced executions of all those subcontracts, but different chains could feature the "most-advanced" executions of different subcontracts.

Taking this into account, we define the coherence relation \sim_4 as follows:

Definition 1 (Coherence, simplified).

$$\begin{split} R^x \sim_{\pmb{A}} R^{||} &::= \\ & \textit{maxFrontier}(R^x) = \bigsqcup_{\mathbb{B} \in \mathcal{B}} \textit{maxFrontier}(R^{||}, \mathbb{B}) \\ & \land \forall \kappa \in \textit{maxFrontier}(R^x) : \mathcal{A}_{\kappa}(R^x) = \mathbb{AU}_{\kappa}^{\pmb{A}}(R^{||}) \\ & \cup (\mathcal{A}_{\kappa}(R^{||}) \cap \{\pmb{A}\}) \end{split}$$

Here, $maxFrontier(R^x)$ denotes the $BitML^x$ contracts (represented by their identifiers κ) in the last configuration of run R^x . Similarly, maxFrontier($R^{||}, \mathbb{B}$) denotes the $BitML^x$ contracts, for which configuration elements $\langle C, \pi \rangle_{\kappa, x}^{\mathbb{B}}$ are available in the last configuration of the intermediate run $R^{||}$. So, intuitively, maxFrontier($R^{||}$, \mathbb{B}) denotes how far the *BitML* execution of the compiled contract has advanced on chain B. Since maxFrontier($R^{||}, \mathbb{B}_i$) may differ across different chains \mathbb{B}_i , we take the supremum over maxFrontier $(R^{||}, \mathbb{B}_i)$ for all chains to obtain a representation of the most advanced state of all subcontracts across all chains. For R^x to be coherent with $R^{||}$, it should match this most advanced state. The second requirement reflects that the authorizations (for guarded contracts κ) provided on runs R^x and $R^{||}$ should match in that the users that authorized κ in \mathbb{R}^x (denoted by $\mathcal{A}_{\kappa}(R^x)$) coincide with the users that authorized κ on all (not yet compensated) blockchains in $R^{||}$ (denoted by $\mathbb{AU}_{\kappa}^{A}(R^{||})$). Here, an exception is made for authorizations of the honest user A who may have only authorized κ in a single chain in $R^{||}$ (in this case $\mathcal{A}_{\kappa}(R^{||}) \cap \{A\} \neq \emptyset$). The reasoning behind that is that honest users can safely replicate their authorizations on all chains, and hence already an authorization on a single chain serves as an evidence that the honest user will synchronously perform such authorizations in the future.

Intermediate strategies. Using coherence, we can formally define how to obtain an intermediate strategy $\Sigma_A^{||} = \mathcal{S}(\Sigma_A^x)$ from a corresponding $BitML^x$ strategy Σ_A^x . We illustrate the main idea behind the strategy compilation using some exemplary compilation rules:

$$\begin{split} R^x \sim_{\pmb{A}} R^{||}|t & \alpha(\kappa) \in \Sigma_{\pmb{A}}^x(R^x) \\ \alpha \in \{dwithdraw, split, reveal\} & \langle C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma_{R^{||}} \\ \underline{\pi.status} = \mathbf{Choice} & \pmb{A} \notin \mathcal{R}_{\kappa}^{ss}(R^{||}) & t < \pi.time \\ & (\pmb{A}: s_{\kappa}^{\pmb{A}}) \in \Sigma_{\pmb{A}}^{||}(R^{||}) \end{split}$$

This rule captures the case where strategy Σ_A^x schedules a (high-priority) move $\alpha(\kappa)$ on a contract κ in R^x . Given a coherent intermediate run $R^{||}$ $(R^x \sim_A R^{||})$, $\Sigma_A^{||}$ will mimic the move by revealing step secret s_κ^A . Revealing the step secret initiates the high-priority move of A for contract κ . Once the step secret is revealed, the strategy $\Sigma_A^{||}$ will

continue scheduling the corresponding κ moves synchronously on all blockchains. By requiring that the current execution time (denoted by t) of the intermediate run $R^{||}$ is smaller than the timeout of contract κ (as captured in the $\pi.time$ field), A can be sure to have sufficient time to perform the consecutive κ moves on all chains. Note that the requirement $\pi.status =$ Choice ensures that C is in an idle state (e.g., has not yet been moved by another user).

Most other rules defining $\Sigma_A^{||}$ are independent of Σ_A^x , since they are concerned with maintaining a synchronized execution state. E.g., the following rule mandates $\Sigma_A^{||}$ to schedule a *ileft* move (corresponding to perform the top-level reveal step in the compiled contract with the step secret) on all blockchains $\mathbb B$ whenever A's step secret has been revealed (indicated by $A \in \mathcal R_\kappa^{ss}(R^{||})$):

$$\frac{A \in \mathcal{R}_{\kappa}^{ss}(R^{||}) \qquad R^{||} \xrightarrow{ileft(\kappa, \mathbb{B})}}{ileft(\kappa, \mathbb{B}) \in \Sigma_{A}^{||}(R^{||})}$$

Similarly, $\Sigma_A^{||}$ will eagerly perfom *right* moves (corresponding to entering a contract κ 's compensation phase) whenever possible:

$$\frac{R^{\mid\mid} \xrightarrow{right(\kappa,\mathbb{B})}}{right(\kappa,\mathbb{B}) \in \Sigma_{A}^{\mid\mid}(R^{\mid\mid})}$$

A full specification of $\Sigma_A^{||}$ can be found in Appendix E.

VI. COMPILER CORRECTNESS

In this section, we sketch the final compiler correctness result and its proof.

Soundness. The core of our compiler correctness proof consists of establishing a general soundness result, essentially stating that for every intermediate run $R^{||}$ resulting from an execution with $\Sigma_A^{||} = \mathcal{S}(\Sigma_A^x)$, we can find a coherent run R^x resulting from an execution with Σ_A^x .

This result ensures that the contracts in $R^{||}$ execute synchronously, in the sense that they do not *diverge* (taking the high-priority choice in one chain \mathbb{B}_i , while taking the low-level branch in another chain \mathbb{B}_j). Such diverging contract executions could never be matched by a single coherent R^x .

For stating soundness, we need to consider that given that Σ_A^x does not aim to schedule a high-priority move, $\Sigma_A^{||} = \mathcal{S}(\Sigma_A^x)$ eagerly proceeds to the compensation phase and, if no compensation is possible, to the low-priority move. We reflect this behavior as a requirement on the $BitML^x$ strategy to be eager, meaning that whenever a move on contract κ in a run R^x is possible, then Σ_A^x should schedule a move related to κ (e.g., performing an authorization, revealing a secret required by the contract or perform a move $\alpha(\kappa)$).

Lemma 1 (Soundness, simplified). Let A be an honest user with an eager $BitML^x$ strategy Σ_A^x and an intermediate semantics strategy $\Sigma_A^{\parallel} = \mathcal{S}(\Sigma_A^x)$. Then

$$\forall R^{||} \ s.t. \ \Sigma_{\textcolor{red}{A}}^{||} \models R^{||}, \ \exists R^x : \Sigma_{\textcolor{red}{A}}^x \vdash R^x \land R^x \sim_{\textcolor{red}{A}} R^{||}$$

Proving soundness requires establishing a multitude of invariants on the configurations of $R^{||}$. In particular, we need to show that as time progresses, the execution of subcontracts whose timeouts are reached are finalized, meaning that they must have advanced synchronously on all chains \mathbb{B}_i , or they got compensated on all blockchains \mathbb{B}_j that did not match eventual high-priority moves before reaching the subcontract-specific timeout. The full proof can be found in Appendix H.

Liquidity. Soundness ensures the existence of a coherent run R^x for every intermediate run $R^{||}$. However, such run R^x with $R^x \sim_A R^{||}$ intuitively only constitutes an execution state, which $R^{||}$ should be guaranteed to match *eventually*. Further, for our final correctness result, we also want to show that the compensation mechanism indeed makes sure that an honest user will always receive adequate compensation. To this end, we want to compare the amount of money that an honest user A obtained (on all blockchains) from running a compiled contract using $\Sigma_A^{||} = \mathcal{S}(\Sigma_A^x)$ with the amount of money A would have obtained from running Σ_A^x .

We make use of the fact that we can show Σ_A^x and $\Sigma_A^{||}$ to be *liquid*, meaning that these strategies will be able to cash out any $BitML^x$ or compiled contract, respectively. Intuitively, this is because the structure of $BitML^x$ contracts ensures that the contract execution cannot get stuck (given an eager honest user strategy): If a malicious user blocks a high-priority choice, the contract can always (unconditionally) advance to the low-priority branch until reaching the default case, which is an unconditional withdraw of funds. This structure is reflected in the compiled contracts.

More formally, a $BitML^x$ strategy is considered liquid, if it can complete any run R^x with $\Sigma_A^x \vdash R^x$ such that its final configuration does not contain any active contracts anymore. We call R^x liquidated in this case. A similar notion can be established for intermediate strategies $\Sigma_A^{||}$. To show the liquidity of a $BitML^x$ strategy Σ_A^x (or an intermediate strategy, respectively), we need to ensure that Σ_A^x does not keep on advertising an infinite amount of new contracts, hence hindering the termination of runs. We formally capture this requirement by requiring the contracts advertised by Σ_A^x to be bounded by a finite set of contract identifiers \mathcal{K} .

With this, we can prove the following result:

Lemma 2 (Intermediate Security). Let A be an honest participant and K be a set of contract identifiers. Let Σ_A^x be an eager strategy of A bounded by K and $\Sigma_A^{||} = S(\Sigma_A^x)$ its corresponding intermediate strategy. Then for every run $R^{||}$ with $\Sigma_A^{||} \models R^{||}$ there exists an extension $\hat{R}^{||} = R^{||} \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$ such that

- 1) $\forall i \in 1 \dots n : \alpha_i \in \Sigma_A^{||}(R^{||} \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{i-1}})$
- 2) $\hat{R}^{||}$ is liquidated

and there exists \hat{R}^x such that $\sum_{A}^{x} \vdash \hat{R}^x$, \hat{R}^x is liquidated and

$$\forall \mathbb{B} \in \mathcal{B} : \mathsf{payout}^{\mathbb{B}}_{\mathbf{A}}(\hat{R^{||}}) - \mathsf{inputs}^{\mathbb{B}}_{\mathbf{A}}(\hat{R^{||}}) \geq \mathsf{payout}^{\mathbb{B}}_{\mathbf{A}}(\hat{R^{x}}) - \mathsf{inputs}^{\mathbb{B}}_{\mathbf{A}}(\hat{R^{x}})$$

This lemma states that the compiled user strategy $\Sigma_A^{||} = \mathcal{S}(\Sigma_A^x)$ can complete any run $R^{||}$ with $\Sigma_A^{||} \models R^{||}$ to a liquidated

run $\hat{R}^{||}$ such that user A on all blockchains \mathbb{B} earns at least as much as they would earn in a corresponding (liquidated) run \hat{R}^x with $\Sigma_A^x \vdash \hat{R}^x$. Here, $\mathsf{payout}_A^\mathbb{B}(\hat{R}^{||})$ denotes the amount of money that gets assigned to A on chain \mathbb{B} during the run $\hat{R}^{||}$ (due to regular payouts or compensations) and $\mathsf{inputs}_A^\mathbb{B}(\hat{R}^{||})$ denotes the inputs of A on \mathbb{B} locked into contracts during $\hat{R}^{||}$. Similarly, $\mathsf{payout}_A^\mathbb{B}(R^{||}) - \mathsf{inputs}_A^\mathbb{B}(\hat{R}^{||})$ denotes the money assigned to A on \mathbb{B} during \hat{R}^x and $\mathsf{inputs}_A^\mathbb{B}(\hat{R}^x)$ the inputs of A on B locked into contracts during \hat{R}^x .

VII. DISCUSSION

In this section, we discuss the rationale for some of the design decisions of $BitML^x$ and the limitations of our approach, hence identifying venues for future work.

Syntax: $BitML^x$ vs BitML. We have made two simplifications in the syntax of $BitML^x$ when compared to that of BitML: (1) We merged BitML's choice operator (+) and time lock mechanism (after) into the priority choice operator (\Rightarrow); and (2) we removed BitML's support for volatile deposits. In the following, we discuss the motivation behind these design choices and their implications on the expressiveness of $BitML^x$.

Priority Choices. $BitML^x$ syntax supports neither the simultaneous choice + nor the user-defined timelock after operators from BitML. Instead, $BitML^x$ syntax supports the operator \Rightarrow , indicating the precedence of moves to the left over moves to the right side of the operator. So, a $BitML^x$ contract is of the form $C := D_1 \Rightarrow (D_2 \Rightarrow (\dots \Rightarrow D_k)\dots)$ where the lowest priority option D_k needs to be a withdraw statement serving as the contract's fallback case that will be executed if none of the higher priority moves were taken. Intuitively, this design ensures that every $BitML^x$ contract is liquid, meaning that funds can never be indefinitely locked in a contract.

Introducing priority choices (as opposed to synchronous choices) in $BitML^x$ is both a necessity and a feature:

Explicit precedence is what allows our compiler to synchronize the contracts in many chains by executing choices in rounds. Further, as outlined in the previous section, liquidity is crucial for proving compiler correctness.

On the other side, liquidity is a desirable contract security feature [11], which $BitML^x$ (as opposed to BitML) enforces by design. Further, we argue that priority choices suitably abstract the combined usage of explicit timelocks and synchronous choices in a way that facilitates easier and more secure smart contract design.

Due to $BitML^x$'s (as well as BitML's) strong adversarial execution model, contracts $D_1 + D_2$ that enable synchronous choices can easily lead to situations where both moves D_1 and D_2 are enabled, and hence the attacker can freely choose which move to take.

To avoid such situations, in many practical use cases, BitML's time lock mechanism (after) serves exactly for implementing precedence between two possible moves: In a BitML contract of the form D_1 + after t: D_2 the

clause after $t: D_2$ effectively lowers the priority of D_2 with respect to D_1 since before time t, contract D_1 can be executed exclusively, enabling the honest user to make this move without interference by the attacker. For implementing precedence, the concrete choice of t does not matter as long as it gives users sufficient time to execute D_1 if desired. The \Rightarrow operator, hence, suitably abstracts the usage of after in those scenarios, relieving the developer from its manual implementation (including the choice of safe values for t).

One drawback of replacing explicit time locks (as supported by after) with priority choices is that the contract developer can only establish a relative execution order but not schedule executions for concrete time points. For specific use cases, contract developers may like to set t to a concrete point in time (e.g., to denote a concrete expiration date). Such a feature is currently not supported by BitMLx. In principle, it would be possible to integrate a corresponding language feature into the BitML^x language, e.g., one could consider adding a timed priority choice operator, where the user explicitly sets the time lock that will be used for compiling this priority choice. However, such a language feature would need to be subject to well-formedness checks ensuring that the user-defined timelock does not interfere with the timelocks automatically inferred by the compiler. Adding such a language feature would consequently make it necessary to expose the user to the implementation details of the compiler. Even worse, for reflecting user-defined timelocks, the BitML^x semantics would need to be extended with an explicit model of time. This model of time would, in turn, make the time-specific execution details of (non-timed) priority choices explicit since otherwise it would not be possible to faithfully model the interference between timed and non-timed priority choices.

Volatile Deposits. BitML supports volatile deposits, denoting deposits from users that can be funded only at the time where the contract needs them. Instead, $BitML^x$ in its current form does not support volatile deposits. Every deposit in $BitML^x$ requires that every other user includes a collateral equal to the amount of such deposit, so that the honest user can be compensated if needed.

If *BitML*^x were to support volatile deposits, this would require that every user also includes additional collateral for those deposits. But at the point in time when the volatile deposit is consumed by the contract, it cannot be guaranteed that all contract users will have sufficient funds to provide the required collateral. As a consequence, it is hard to enforce the synchronicity of such an execution step on all chains. This issue could be circumvented by requiring all contract users to provide the collaterals for all eventual volatile deposits at the contract stipulation time. However, such an implementation would defeat the original purpose of volatile deposits, which is to avoid binding too many financial resources in a contract for the whole duration of the contract execution.

System assumptions and supported compilation targets. $BitML^x$ compiles to BitML smart contracts, which in turn can be translated to transactions in Bitcoin-like cryptocurrencies.

We consider cryptocurrencies Bitcoin-like if they are UTXObased and support a scripting language that subsumes the one of Bitcoin. More precisely, such scripting languages need to be able to restrict the spending of unspent transaction outputs (UTXOs) based on conditions that include (i) signature checks (providing signatures that correctly verify for prespecified public keys); (ii) hash locks (providing preimages for predefined hash values); (iii) time locks (requiring a minimal system time has been reached); (iv) the evaluation of conditions (on input values) as defined in Fig. 4. Examples of such cryptocurrencies are sharing the Bitcoin scripting language (such as Dogecoin [12], Bitcoin Cash [13] or Litecoin [14]), or more expressive scripting languages (e.g., Cardano [15] supports an extended UTXO model featuring a Turing-complete scripting language). Account-based cryptocurrencies (such as Ethereum [16], Solana [17], Ripple [18], or Stellar [19]), which explicitly track user and smart contract balances, or currencies with more limited scripting support (such as Monero [20], Zerocoin [21], or Zerocash [22]) are currently not in scope.

More practically, for compiling a $BitML^x$ contract to k blockchains $\mathbb{B}_1, \ldots, \mathbb{B}_k$, it is sufficient to ensure that the BitML compiler supports the chains $\mathbb{B}_1, \ldots, \mathbb{B}_k$ as compilation targets. Fortunately, the existing implementation of the BitML compiler outputs an abstract transaction format (called Balzac [23], [24]). Hence, it suffices to provide serializations from Balzac to transactions in the desired target blockchains $\mathbb{B}_1, \ldots, \mathbb{B}_k$. In particular, since our compilation targets BitML, this means that our compiler correctness result is independent of the choice of targeted blockchains.

Expanding the scope of $BitML^x$ beyond Bitcoin-like cryptocurrencies could be done by encoding Balzac transactions and emulating their execution within the targeted system. This could be a viable path towards extending $BitML^x$ to account-based cryptocurrencies with (quasi) Turing-complete contract support (such as Ethereum or Solana).

Assumptions on honest users. The execution of a compiled $BitML^x$ contract assumes that honest users are online during the execution of the contract, monitoring the potentially k blockchains to attempt to include the transactions corresponding the next move when needed. In practice, the honest user would also need to pay a fee for each of such moves, since the inclusion of the corresponding transaction in the i-th blockchain requires a transaction fee. Relaxing these practical assumptions on honest users is an interesting future research direction, for instance, by compiling $BitML^x$ contracts into off-chain solutions such as payment channels or rollups [25].

Required collateral. For simplicity, our solution uses an overly conservative formula for calculating the collateral needed in a $BitML^x$ contract. Through static analysis of contract code, upper bounds for the best-case withdraw output for each user can be determined, and the needed collateral could be reduced to the sum of those upper bounds for all users. An additional optimization in this direction could include allowing users to withdraw part of their collateral as

the contract progresses and these upper bounds are lowered.

Supported applications. Since $BitML^x$ relies on BitML as its compilation target, $BitML^x$ inherits known limitations of the BitML language. In particular, $BitML^x$ (as BitML) only supports smart contracts involving a fixed set of users, and its computational expressiveness is limited (e.g., both BitML and $BitML^x$ do not support recursion). This excludes applications that are open-ended (such as payment channels) or that rely on dynamically changing sets of users (such as general decentralized exchanges or crowdfunding). Recent extensions of the BitML language [26] allow for expressing a limited form of recursion by enabling contract users to renegotiate parts of the contract during execution. It remains interesting future work to investigate whether $BitML^x$ can be extended to support this feature as well.

The simplifications of $BitML^x$ with respect to BitML (omission of explicit time locks and volatile deposits) exclude applications that either (i) rely on payments that shall be scheduled for a specific time in the future (e.g., modeling loans that shall be returned at a specific date); or (ii) enable users to top up the contract funds during execution (e.g., to pay a mediator to resolve conflicts on demand).

Despite these limitations, $BitML^x$ can still express many interesting cross-chain applications, as we will illustrate in the following section.

VIII. APPLICATIONS

Here, we overview cross-chain applications and express them with $BitML^x$ language. We then evaluate the resources required to execute them in the underlying cryptocurrencies.

Multichain donations. Several organizations such as non-profit, charities, universities, faith-based or mission-driven organizations accept cryptocurrency donations in different cryptocurrencies to support their cause [27]. For instance, organizations in [27] accept donations in over 80 different cryptocurrencies. Users might have a varied portfolio of cryptocurrencies, and they can decide to donate in different denominations. In the following, we show how to use $BitML^x$ to implement several instances of such use case, depending on how the denomination is chosen.

First, the receiver can select the denomination of the donation, with the $BitML^x$ contract shown in Fig. 7. Here, the donor A offers to donate either 1 or 1 to B, who can then decide the denomination by exercising the priority choice between PayBTC and PayDGC. Second, both participants can jointly decide the denomination of the donation by means of a coin flip, as shown in Fig. 7. Here, A commits to a secret bit x and B commits to a secret bit y. If both are heads or tails, the donor pays 1. Otherwise, the donor pays 1.

Multichain payments with exchange service. As with fiat currencies, cross-cryptocurrency payments often require an intermediate exchange service that performs the exchange between different cryptocurrencies. Next, we illustrate how $BitML^x$ can be used to implement such scenarios (c.f. Fig. 7).

Assume that a customer C only holds bitcoins and goes to a restaurant (i.e., R). Further assume that R's portfolio

```
\{A: 1\cline{0}|A: 1\cline{0}\} Donate
        Donate
                       = PayBTC \Rightarrow PayDGC \Rightarrow Refund
        PayBTC
                       = B : [withdraw(1B, 0D) \rightarrow B,
                               withdraw(0\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ ) \to A
                       = [withdraw(0B, 1D) \rightarrow B,
        PayDGC
                          withdraw(1B, 0D) \rightarrow A
                       = withdraw(18, 10) \rightarrow A
        Refund
\{A: 1B | A: 1D\} Donate Agreed
DonateAgreed = CoinToss \Rightarrow Refund
CoinToss = reveal^* x.(WinBTC \Rightarrow WinDGC)
WinBTC = reveal^* y \land |x| = |y| \cdot (PayBTC \Rightarrow PayDGC)
WinDGC = reveal^* y \land |x| \neq |y| \cdot (PayDGC \Rightarrow PayBTC)
PayBTC =
                          [withdraw(1B, 0D) \rightarrow B,
                          withdraw(0\cRed{B}, 1\cRed{D}) \rightarrow A
PayDGC =
                          [withdraw(0B, 1D) \rightarrow B,
                          withdraw(1B,0D) \rightarrow A
Refund =
                          withdraw(1\Breve{B}, 1\Breve{D}) \rightarrow A
     \{C: 10\ | X: 100\ | \} Exchange Service
    ExchangeService = Payment \Rightarrow RefundAll
    Payment = DirectPay \Rightarrow ExchangePay
                              =C:[\text{withdraw}(10\cRed{0},0\cRed{0})
ightarrow R,
    DirectPay
                                       withdraw(0\cRed{0}\cRed{0}, 100\cRed{0}) 
ightarrow X]
                               ExchangePay
                                 withdraw((0B, 100D) \rightarrow R]
                               = [\text{withdraw}(10^{\circ}, 0^{\circ})] \rightarrow C,
    RefundAll
                                  withdraw(0^{1}_{0}, 100^{1}_{0}) \rightarrow X
   \{B: 3B | L: 30D \} Loan
   Loan = ExecLoan \Rightarrow RefundLoan
   ExecLoan
                         = split [withdraw (0^{\circ}_{\bullet}, 30^{\circ}_{\bullet}) \rightarrow B,
                         (3B, 0D) \rightarrow Installment[3]
   Installment[i]
                         = M : \text{split} [\text{withdraw}(1B, 0D)] \rightarrow B,
                         (i-1\beta,0\beta) \rightarrow Installment[i-1]
                          +> withdraw(i \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ ) \rightarrow L
                         = [withdraw(3B, 0D) \rightarrow B,
   RefundLoan
                         withdraw(0B, 30D \rightarrow L]
```

Fig. 7. $BitML^x$ contract for applications (from top to bottom): multichain donation (with receiver chosen denomination), multichain donation (with jointly agreed denomination), multichain payment with exchange service, multichain loan with mediator. For readability, we use reveal* s as a shortcut for reveal s if $0 \le |s| \le 1$.

consist of both bitcoins and dogecoins. In this setting, after the dinner is finished, the restaurant offers the customer C two options for the payment: (i) pay in bitcoins directly to R (i.e., DirectPay); or (ii) pay in dogecoins using a predefined exchange rate $-1\frac{R}{2}:10\frac{R}{2}$ in this example—offered by an exchange service X (i.e., ExchangePay). In the former case, the restaurant R gets paid $10\frac{R}{2}$ while the exchange X gets back $100\frac{R}{2}$. In the latter case, the exchange X receives bitcoins whereas the restaurant R gets the payment in dogecoins.

Multichain loan with mediator. Assume a borrower B and a lender L. The borrower has $3\frac{1}{9}$ and wants to take a loan of $30\frac{1}{9}$. We assume that the exchange rate between bitcoins and dogecoins is $1\frac{1}{9}:10\frac{1}{9}$. Having a surplus of dogecoins, the lender is willing to grant such a loan of $30\frac{1}{9}$. However, since lender and borrower are mutually distrusted, the lender wants to hold the $3\frac{1}{9}$ from the borrower as a collateral for the duration of the loan (or until one of the installment's payment of the loan is missed by the borrower).

In such a setting, a lending protocol works in three stages

TABLE I SUMMARY OF RESOURCES PER BLOCKCHAIN REQUIRED BY $\it BitML^x$ Contracts described in Section VIII.

	Generated Txs	Executed Txs
Donate (Fig. 7)	51	6
DonateAgreed (Fig. 7)	51	6
ExchangeService (Fig. 7)	207	6
Loan (Fig. 7)	4098	7

as follows. In the first stage, the lender and borrower input the loan coins (i.e., $30\mbox{\sc D}$) and the collateral coins (i.e., $3\mbox{\sc B}$) into the contract (i.e., Loan). In such a contract, (i) borrower receives the $30\mbox{\sc D}$ of the loan (i.e., ExecLoan) and the $3\mbox{\sc B}$ of the collateral are locked into a second contract (i.e., Installment), triggering the second stage.

The contract *Installments* works as follows. At each step i, if the borrower has paid $10\rlap.$ 0 to the lender, the mediator M, trusted to monitor when an installment payment of $10\rlap.$ 0 from the borrower to lender occurs, authorizes the release of $1\rlap.$ 1 in favor of the borrower. This second stage of the loan protocol ends when either (i) the last step is finished; or (ii) the borrower does not pay the installment at the i-th step. In the former case, the loan protocol is finished since the borrower returned the complete loan to the lender, getting the complete collateral back in exchange. In the latter case, the lender can claim the amount of bitcoins that remain as collateral as a payment for the default of the borrower.

The application described so far can be implemented with the $BitML^x$ contract shown in Fig. 7. Note that we have made several simplifications of the protocol for ease of exposition. It is possible to extend the $BitML^x$ contract to model other practical features such as (i) different amounts for the loan and the collateral to account for exchanges rates different to $1\frac{11}{15}:10\frac{11}{15}$; (ii) different amounts for each of the installment payments; (iii) consider a third blockchain where the mediator M gets paid by the borrower B and the lender L if the mediator's task in the loan is successfully carried out.

Evaluation. Now, we discuss the blockchain resources required to execute the $BitML^x$ contracts presented so far in this section. For that, we have compiled the $BitML^x$ contracts into BitML contracts using our compiler (Section V), whose source code is available at [10]. Then, using the BitML compiler in [1], we have transformed the BitML contracts into Bitcoin transactions. As a result, we have obtained the total number of transactions required in Bitcoin to encode the complete logic of each application, as shown in Table I. However, not all transactions need to be included in the blockchain, only those required by the user strategies. Even in the worst case, this corresponds to the depth of the execution tree, which is considerably smaller (see Table I).

IX. RELATED WORK

Existing approaches in the literature [4] for cross-chain applications, which all focus on custom designs tailored to the specific application, can be grouped into (i) migration

protocols or bridges; and (ii) cryptographic protocols. Next, we overview and compare them with our work.

Migration protocols or bridges. The cornerstone of this protocol class is to rely on a coordinator to (i) lock funds in the different cryptocurrencies as required by the cross-chain smart contract; (ii) release these funds into a (possibly different) cryptocurrency with support for expressive smart contracts (e.g., Ethereum) that it is in then used to execute the smart contract; and (iii) redistribute the final distribution of funds back to the individual cryptocurrencies, as indicated by the final state of the cross-chain smart contract. The role of the coordinator is implemented by trusted hardware module [28] or a committee of users that jointly authorize the back and forth transfer of funds across cryptocurrencies [29].

The security of this approach is built upon an additional trust assumption on the coordinator itself, either in the form of honest majority across the committee or the trust on the hardware module itself. Moreover, this approach requires a cryptocurrency with support for stateful smart contracts, limiting its applicability to cryptocurrencies like Bitcoin.

Custom cross-chain cryptographic protocols. Alternatively to migration protocols or bridges, this group of works contributes an application-dependent cryptographic protocol as a synchronization overlay among the cryptocurrencies involved in the cross-chain application. Examples of these protocols exist for payment channels [30], payment channel networks [31], coin mixing [7], oracle-based contracts [32], and beyond.

While this approach departs from the additional trust assumption as in migration protocols, the design of these custom cryptographic protocols requires substantial (cryptographic) expertise. Moreover, the security of these protocols is proven from scratch, requiring involved proofs in complex cryptographic proof frameworks, an error-prone task as demonstrated so far in the literature [5].

Different to existing approaches, in our work we provide a domain-specific language that can be used to describe the functionality of the cross-chain smart contract, abstracting away the cryptographic details of the protocol realizing the contract execution. The $BitML^x$ smart contract is then compiled into several contracts (i.e., one per involved cryptocurrency) that are compatible with Bitcoin-like cryptocurrencies and a strategy for each honest user on how to interact with them. Our approach comes with formally proven correctness guarantees: If an honest user follows the prescribed strategy when interacting with the compiled smart contract at each cryptocurrency, they are guaranteed to end up with at least as many funds as described by the execution of the $BitML^x$ cross-chain contract.

X. CONCLUSION

We present $BitML^x$, the first domain-specific language for cross-chain smart contracts. We show a compiler to automatically translate a $BitML^x$ contract into one contract per involved cryptocurrency and a user strategy to interact with them, and prove that a user following such strategy will end

up with as many funds as in the corresponding execution of the $BitML^x$ contract. We have implemented the $BitML^x$ compiler and showcased its utility in the design of illustrative examples of cross-chain applications, e.g., multichain loans.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their helpful feedback. Furthermore, we would like to thank Minhua Liu for assisting in the preparation of the paper artifacts and helping improve the paper presentation. This work has been supported by the Heinz Nixdorf Foundation through a Heinz Nixdorf Research Group (HN-RG) and funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy—EXC 2092 CASA—390781972. Further, this work has been partially supported by the ESPADA project (grant PID2022-142290OB-I00), MCIN/AEI/10.13039/501100011033/ FEDER, UE; and by the PRODIGY project (grant ED2021-132464B-I00), funded by MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/ PRTR.

REFERENCES

- M. Bartoletti and R. Zunino, "Bitml: A calculus for bitcoin smart contracts," Cryptology ePrint Archive, Paper 2018/122, 2018, https://eprint.iacr.org/2018/122. [Online]. Available: https://eprint.iacr.org/2018/122
- [2] S. A. Thyagarajan, G. Malavolta, and P. Moreno-Sanchez, "Universal atomic swaps: Secure exchange of coins across all blockchains," in Symposium on Security and Privacy (SP). IEEE, 2022, pp. 1299–1316.
- [3] M. Herlihy, "Atomic cross-chain swaps," in Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing (PODC), 2018.
- [4] A. Zamyatin, M. Al-Bassam, D. Zindros, E. Kokoris-Kogias, P. Moreno-Sanchez, A. Kiayias, and W. J. Knottenbelt, "Sok: Communication across distributed ledgers," in *Financial Cryptography and Data Security*, 2021, pp. 3–36.
- [5] E. Tairi, P. Moreno-Sanchez, and C. Schneidewind, "Ledgerlocks: A security framework for blockchain protocols based on adaptor signatures," in *Conference on Computer and Communications Security (CCS)*, 2023, pp. 859–873.
- [6] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, "Anonymous multi-hop locks for blockchain scalability and interoperability," in *Network and Distributed Systems Security (NDSS)* Symposium, 2019.
- [7] N. Glaeser, M. Maffei, G. Malavolta, P. Moreno-Sanchez, E. Tairi, and S. A. K. Thyagarajan, "Foundations of coin mixing services," in *Conference on Computer and Communications Security*, 2022, pp. 1259–1273.
- [8] P. Gerhart, D. Schröder, P. Soni, and S. A. K. Thyagarajan, "Foundations of adaptor signatures," in *International Conference on the Theory* and Applications of Cryptographic Techniques, ser. Lecture Notes in Computer Science, vol. 14652, 2024, pp. 161–189.
- [9] J. Harris and A. Zohar, "Flood & loot: A systemic attack on the lightning network," CoRR, vol. abs/2006.08513, 2020. [Online]. Available: https://arxiv.org/abs/2006.08513
- [10] F. Badaloni, S. Holler, C. Oikonomou, P. Moreno-Sanchez, and C. Schneidewind, "Bitmlx compiler," https://github.com/hn-rg/BitMLx.
- [11] M. Bartoletti and R. Zunino, "Verifying liquidity of bitcoin contracts," in International Conference on Principles of Security and Trust. Springer, 2019, pp. 222–247.
- [12] "Dogecoin whitepaper v1.1," https://dogechain.dog/DogechainWP.pdf.
- [13] "Bitcoin cash website," https://bitcoincash.org.
- [14] "Litecoin website," https://litecoin.org.

- [15] M. M. T. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, M. P. Jones, and P. Wadler, "The extended UTXO model," in Financial Cryptography and Data Security FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers, ser. Lecture Notes in Computer Science, M. Bernhard, A. Bracciali, L. J. Camp, S. Matsuo, A. Maurushat, P. B. Rønne, and M. Sala, Eds., vol. 12063. Springer, 2020, pp. 525–539. [Online]. Available: https://doi.org/10.1007/978-3-030-54455-3_37
- [16] V. Buterin, "Ethereum whitepaper," https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf.
- [17] A. Yakovenko, "Solana: A new architecture for a high performance blockchain," https://solana.com/solana-whitepaper.pdf.
- [18] B. Chase and E. MacBrough, "Analysis of the XRP ledger consensus protocol," CoRR, vol. abs/1802.07242, 2018. [Online]. Available: http://arxiv.org/abs/1802.07242
- [19] D. Mazieres, "The stellar consensus protocol:
 A federated model for internet-level consensus," https://stellar.org/learn/stellar-consensus-protocol.
- [20] S. Noether and A. Mackenzie, "Ring confidential transactions," Ledger, vol. 1, pp. 1–18, 2016. [Online]. Available: https://ledgerjournal.org/ojs/index.php/ledger/article/view/34
- [21] I. Miers, C. Garman, M. Green, and A. D. Rubin, "Zerocoin: Anonymous distributed e-cash from bitcoin," in 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013. IEEE Computer Society, 2013, pp. 397–411. [Online]. Available: https://doi.org/10.1109/SP.2013.34
- [22] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014. IEEE Computer Society, 2014, pp. 459–474. [Online]. Available: https://doi.org/10.1109/SP.2014.36
- [23] N. Atzei, M. Bartoletti, S. Lande, and R. Zunino, "A formal model of bitcoin transactions," in *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Meiklejohn and K. Sako, Eds., vol. 10957. Springer, 2018, pp. 541–560. [Online]. Available: https://doi.org/10.1007/978-3-662-58387-6_29
- [24] "Balzac: A domain-specific language to write bitcoin transactions," https://github.com/balzac-lang/balzac.
- [25] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, "Sok: Layer-two blockchain protocols," in *Financial Cryptography and Data Security (FC)*, vol. 12059, 2020, pp. 201–226.
- [26] M. Bartoletti, M. Murgia, and R. Zunino, "Renegotiation and recursion in bitcoin contracts," in Coordination Models and Languages 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings, ser. Lecture Notes in Computer Science, S. Bliudze and L. Bocchi, Eds., vol. 12134. Springer, 2020, pp. 261–278. [Online]. Available: https://doi.org/10.1007/978-3-030-50029-0_17
- [27] T. G. Block, "Donate to your favorite cause," https://thegivingblock.com/donate/.
- [28] I. Bentov, Y. Ji, F. Zhang, L. Breidenbach, P. Daian, and A. Juels, "Tesseract: Real-time cryptocurrency exchange using trusted hardware," in *Conference on Computer and Communications Security (CCS)*, 2019, pp. 1521–1538.
- [29] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song, "zkbridge: Trustless cross-chain bridges made practical," in Conference on Computer and Communications Security (CCS), 2022, pp. 3003–3017.
- [30] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostáková, M. Maffei, P. Moreno-Sanchez, and S. Riahi, "Generalized channels from limited blockchain scripts and adaptor signatures," in *Conference on the Theory* and Application of Cryptology and Information Security (AsiaCrypt), vol. 13091, 2021, pp. 635–664.
- [31] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, "Anonymous multi-hop locks for blockchain scalability and interoperability," in *Network and Distributed System Security Symposium* (NDSS), 2019.
- [32] V. Madathil, S. A. K. Thyagarajan, D. Vasilopoulos, L. Fournier, G. Malavolta, and P. Moreno-Sanchez, "Practical decentralized oracle

contracts for cryptocurrencies," *IACR Cryptol. ePrint Arch.*, p. 499, 2022. [Online]. Available: https://eprint.iacr.org/2022/499

15

APPENDIX A $BitML^x$ SYNTAX

A. Preconditions

The $BitML^x$ syntax tries to remain as close to it as possible to the single chain version of BitML. Notable changes are:

- Persistent deposits take balance, which is a list of currencies.
- No support for volatile deposits
- We add a new precondition for establishing contract start time unique initial id. This is limited syntactically to a single precondition of this kind.

```
G ::= \\ \mid A : ! \ B @ \vec{x} \\ \mid A : \text{secret } s \\ \mid G | \ G' \\ G \mid (t_0, \kappa_0) \\ B = [v_1 \mathbb{B}_1, \dots, v_k \mathbb{B}_k] \quad \text{Balance}
```

Preconditions are commutative.

B. Contracts

- The contract sum has been replaced with priority choices.
- Guarded contracts are only allowed on the left side of a priority choice.
- withdraw can also be used as a top-level contract and acts as a terminal constructor. It has also been generalized to
 map users to the corresponding funds they withdraw on each blockchain.
- We remove the put statement because again, we don't support volatile deposits.
- We also remove the after statement because our timed semantics will be coupled to our choice operator.
- We write signature authorizations with lists of signing users. Technically, concatenating multiple single-user authorizations is still allowed by the syntax.
- split now takes a vector of balances to split for each contract. withdraw also has a list of balances for each user.

```
\begin{array}{l} C := D \Leftrightarrow C & \text{Top-level contracts} \\ | \, \text{withdraw} \stackrel{\vec{\mathbf{B}}}{\to} \stackrel{\vec{A}}{A} & \\ D := \, \text{withdraw} \stackrel{\vec{\mathbf{B}}}{\to} \stackrel{\vec{A}}{A} & \text{Guarded contracts} \\ | \, \text{split} \stackrel{\vec{\mathbf{B}}}{\to} \stackrel{\vec{C}}{C} & \\ | \, \stackrel{\vec{A}}{:} D & \\ | \, \text{reveal} \stackrel{\vec{s}}{\:} \text{if} \, p \, \text{then} \, C & \end{array}
```

C. WellFormdness Check

The well-formedness check further constraints what kind of $BitML^x$ contract advertisements are actually valid. It rules out things like inconsistent funds distribution on splits and withdraws or references to non-existing participants.

The auxiliary functions we use throughout the rules are defined at the end of the appendix.

```
wellFormed(G, D) := wellFormed(G, D \Rightarrow C, \mathbf{B}(balance(G)))
```

1) Top-Level-Contracts: A top-level contract is well-formed if it is a priority choice between well-formed contracts or if it is a well-formed withdraw (which we explain on the guarded version).

$$\frac{wellFormed(G,C,\mathbf{B}) \quad wellFormedG(G,D,\mathbf{B})}{wellFormed(G,D \Leftrightarrow C,\mathbf{B})} \ WF-PChoice$$

$$\frac{\vec{\mathbf{B}} = \mathbf{B}_1, \dots, \mathbf{B}_k \quad \sum_{1}^k \mathbf{B}_i = \mathbf{B} \quad \forall A_i \in \vec{A}. \ A_i \in users(G)}{wellFormed(G,\mathbf{Withdraw} \ \vec{\mathbf{B}} \to \vec{A},\mathbf{B})} \ WF-Withdraw$$

2) Guarded Contracts:

$$\begin{split} \vec{B} &= \mathbf{B}_{1}, \dots, \mathbf{B}_{k} \\ \vec{C} &= C_{1}, \dots, C_{k} \\ \hline \vec{C} &= C_{1}, \dots, C_{k} \\ \hline \end{bmatrix}_{1}^{k} \mathbf{B}_{i} = \mathbf{B} \\ \forall i \in [1, \dots, k].wellFormed(G, C_{i}, \mathbf{B}_{i}) \\ \hline wellFormedG(G, \operatorname{split} \vec{\mathbf{B}} \to \vec{C}, \mathbf{B}) \\ \hline \\ wellFormedG(G, \operatorname{split} \vec{\mathbf{B}} \to \vec{C}, \mathbf{B}) \\ \hline \\ wellFormedG(G, \operatorname{reveal} \vec{s} \text{ if } p \text{ then } C, \mathbf{B}) \\ \hline \\ wellFormedG(G, \operatorname{peveal} \vec{s} \text{ if } p \text{ then } C, \mathbf{B}) \\ \hline \\ wellFormedG(G, \vec{A}; D, \mathbf{B}) \\ \hline \\ \vec{B} &= \mathbf{B}_{1}, \dots, \mathbf{B}_{k} \\ \hline \\ \vec{B} &= \mathbf{B}_{1}, \dots, \mathbf{B}_{k} \\ \hline \\ wellFormedG(G, \text{withdraw } \vec{v} \to \vec{A}, \mathbf{B}) \\ \hline \\ & \text{APPENDIX B} \\ \\ & \text{BitML}^{x} \text{ Semantics} \\ \end{split}$$

A. Configurations

A configuration Γ encapsulates the complete current state including contract advertisements, active contracts, assigned contracts, committed and revealed (user) secrets and authorizations.

- Active contracts $\langle C, \mathbf{B} \rangle_{\kappa}$ have a generelized balance, expressed as a vector of coins.
- User deposits are explicit about the blockchain they live in.

$$\begin{split} \Gamma &:= \{G\}C & \text{Configurations} \\ & | \ \langle C, \mathbb{B} \rangle_{\kappa} \\ & | \ \langle A, \mathbb{B} \rangle_{\kappa} \\ & | \ A : [\phi] \\ & | \ \{A : s \# N\} \\ & | \ A : s \# N \\ & | \ \Gamma \| \Gamma' \\ \mathbb{B} &= [v_1 \mathbb{B}_1, \dots, v_k \mathbb{B}_k] \\ & \phi ::= \kappa \triangleright D \mid \# \triangleright D \mid \{G\}C \end{split}$$
 Contract balance

B. Guarded contracts

Rules in this section describe how guarded contracts transition. They are always assumed to be the left choice in the context of a priority choice with another contract.

- 1) Withdraw: The D-Withdraw rule says that if:
- We have a withdraw guarded contract called D, where each user in \vec{A} is given a corresponding balance in \vec{B} .
- The values on the assigned balances add up to the contract balance, on each blockchain. Notice that this is expressed in vector arithmetic.
- We have fresh identifiers for the all new user deposits.

Then, when D is on the left side of a priority choice, we can transition by consuming the active contract from the configuration and including new assigned contracts for each participant A_i with balance B_i and identifier κ_i . If there's extra context Γ , then it's conserved.

$$\begin{split} D &= \texttt{withdraw} \ \vec{\mathsf{B}} \to \vec{A} \\ \vec{\mathsf{B}} &= \mathsf{B}_1, \dots, \mathsf{B}_n \\ \vec{A} &= A_1, \dots, A_n \\ \hline \langle D \Leftrightarrow C, \mathsf{B} \rangle_\kappa \mid \Gamma \xrightarrow{dwithdraw(\kappa)} \prod_{i=1}^n \langle A_i, \mathsf{B}_i \rangle_{\kappa_i} \mid \Gamma \end{split} \\ D &= \mathsf{Withdraw} \end{split}$$

- 2) Split: The D Split rule says that if:
- We have a split guarded contract called D, where the contract funds are divided among new contracts \vec{C} with corresponding balances \vec{B} .
- The values on the funds mapping add up to the contract balance, on each blockchain.
- We have fresh identifiers for the new user active contracts.

Then, when D is on the left side of a priority choice, we can transition by removing the original active contract from the configuration and including a new active contract for each C_i with balance B_i and identifiers κ_i . If there's extra context Γ , then it's conserved.

$$D = \operatorname{split} \vec{\mathbf{B}} \to \vec{C}$$

$$\vec{\mathbf{B}} = \mathbf{B}_1, \dots, \mathbf{B}_n$$

$$\vec{C} = C_1, \dots, C_k$$

$$\kappa_1, \dots, \kappa_k \text{ fresh}$$

$$\langle D \Rightarrow C, \mathbf{B} \rangle_{\kappa} \mid \Gamma \xrightarrow{split(\kappa)} \bigvee_{i=1}^k \langle C_i, \mathbf{B}_i \rangle_{\kappa_i} \mid \Gamma$$

- 3) Reveal: The D-Reveal rule says that if:
- We have a reveal guarded contract called D with secrets \vec{s} , predicate p and follow-up contract C'.
- All of the secrets are revealed in Δ .
- The semantics on p when substituting it's free variables by the values revealed in Δ evaluate to true.

Then, when D is on the left side of a priority choice, we can transition to a configuration where we replace the active contract with a new one containing C'.

$$\begin{split} D &= \text{reveal } \vec{s} \text{ if } p \text{ then } C' \\ \vec{s} &= s_1, \dots, s_k & [\![p]\!]_\Delta = true \\ \Delta &= \underset{i=1}{\overset{k}{\parallel}} A: s_i \# N_i \\ \hline \\ \langle D \Rightarrow C, \mathbf{B} \rangle_\kappa \mid \Delta \mid \Gamma \xrightarrow{reveal(\kappa)} \langle C', \mathbf{B} \rangle_{\kappa'} \mid \Delta \mid \Gamma \end{split}$$

The D - Auth rule says that if:

- We have a contract D guarded by the signatures of participants \vec{A} .
- All of the required signatures are provided in Δ .
- An active contract κ including a priority choice with D on the left side in a context Γ can transition to a configuration Γ' .

Then, \vec{A} : D can also do the same transition when including Δ in the context.

$$\Delta = \prod_{\substack{A_i \in \vec{A} \\ A_i \in \vec{A}}} A_i [\kappa \triangleright D]$$

$$\frac{\langle D \Rightarrow C, \mathbf{B} \rangle_{\kappa} | \Gamma \xrightarrow{\alpha} \Gamma'}{\langle \vec{A} \colon D \Rightarrow C, \mathbf{B} \rangle_{\kappa} | \Delta | \Gamma \xrightarrow{\alpha} \Gamma'} D - Auth$$

C. Top-Level Contracts

Rules here describe the transitions to take a right choice, either by skipping to another contract, or finishing with a Withdraw.

1) Withdraw: The transition rule for a top-level withdraw has the same final effect as a guarded withdrawcontract.

$$\begin{split} C &= \text{withdraw } \vec{\mathsf{B}} \to \vec{A} \\ \vec{\mathsf{B}} &= \mathsf{B}_1, \dots, \mathsf{B}_n \\ \vec{A} &= A_1, \dots, A_n \\ \hline \langle C, \mathsf{B} \rangle_\kappa \mid \Gamma \xrightarrow{cwithdraw(\kappa)} \prod_{i=1}^n \langle A_i, \mathsf{B}_i \rangle_{\kappa_i} \mid \Gamma \end{split}$$

2) Skip: The C-Skip rule says that a priority choice can transition to the right side. Restrictions in the the strategies will only allow the attacker to skip when there's consensus for it.

$$\frac{\kappa' \text{ fresh}}{\langle D \Rightarrow C, \mathbf{B} \rangle_{\kappa} \mid \Gamma \xrightarrow{skip(\kappa)} \langle C, \mathbf{B} \rangle_{\kappa'} \mid \Gamma} C - Skip$$

D. Authorizations

These rules don't transform active contracts but instead only add some authorization by some participant in the context, as preconditions for other rules.

- 1) Signatures: The A Auth rule says that if:
- We have an active contract where the left side is guarded with authorizations by \vec{A} .
- A's signature is not present in the configuration.

Then we can transition by adding the authorization in the configuration.

$$\frac{A \in \vec{A}}{\langle \vec{A} \colon \ D \Rightarrow C, \mathbf{B} \rangle_{\kappa} \mid \Gamma \xrightarrow{A:\kappa} \langle \vec{A} \colon \ D \Rightarrow C, \mathbf{B} \rangle_{\kappa} \mid A[\kappa \triangleright D] \mid \Gamma} \ A - Signature$$

2) Secrets: The A-RevealSecret rule says that if A really has the value for a secret they committed, then they can always reveal it. Remember that dishonest users can commit to any secret, even if they don't actually know the value, which is represented by the \bot length.

$$\frac{N \neq \bot}{\{A: a\#N\} \mid \Gamma \xrightarrow{A:a} A: a\#N \mid \Gamma} A - RevealSecret$$

E. Stipulation Protocol

BitMLx stipulation protocol looks very similar to BitML's, but has the option to abort a contract, which destroys the advertisement, commitment and signatures. This design is much simpler than our previous one, which required an extra intermediate step. My reasoning for it is that, just like we can map a compensated contract as a left move, we can map all combinations of not publishing, refunding or compensating to an abort.

- 1) Advertisement: A contract advertisement $\{G\}C \mid \Gamma$ is valid if
- All secrets in the contract are fresh.
- At least one user is honest.

$$users(G) \cap Hon \neq \emptyset$$

$$wellFormed(\{G\}C)$$

$$\forall s \in secrets(G), s \text{ fresh}$$

$$\Gamma \xrightarrow{advertise(G,C)} \{G\}C \mid \Gamma$$

$$S - Advertisement$$

Where:

- secrets(G) is the set of all secrets that appear in the preconditions G.
- users(G) is the set of all participants that have a deposit in G.
- wellFormed(G,C) checks some basic correctness conditions on the contract, such as all secrets used in reveal statements being committed to in the preconditions and all branches on split and withdraw statements add up to the original balance.
- 2) Secret Commitment: This is taken from BitML almost verbatim. Each participant A can commit to the contract advertisement $\{G\}C$ if
 - The contract is advertised in the current configuration.
 - If they are honest, all of their secrets have some length $N \in \mathbb{N}$. If not, the length can also be \bot , which represents a secret for which they don't know the value.
 - They have not committed any of their secrets in G in the past.
 - They now commit to all of their secrets in G.

$$A \in users(G)$$

$$\forall s \in secrets_{A}(G), N \in \begin{cases} \mathbb{N}, & \text{if } A \in Hon \\ \mathbb{N} \cup \{\bot\}, & \text{otherwise} \end{cases}$$

$$\forall s \in secrets_{A}(G), \nexists N : \{A : s \# N\} \in \Gamma$$

$$\Delta = \underset{s \in secrets_{A}(G)}{\parallel} \{A : s \# N_{s}\}$$

$$\frac{A : s \# N_{s}}{\{G\}C \mid \Gamma \xrightarrow{commit(A, \{G\}C)}\}} \{G\}C \mid \Delta \mid A [\# \triangleright \{G\}C] \mid \Gamma$$

$$S - AuthCommit$$

Where:

- $secrets_A(G)$ is the set of all secrets in G that A commits to.
- 3) Stipulation Authorization: Each participant A can authorize an advertisement if everyone has committed to their secrets. This corresponds in the low levels to authorizing their deposits to be spent in favor of the contract.

$$\frac{\forall A_i \in users(G). \ A_i[\# \triangleright \{G\}C] \in \Gamma}{\{G\}C \mid \Gamma \xrightarrow{sign(A,\{G\}C)} \{G\}C \mid A[\{G\}C] \mid \Gamma} \ S - AuthInit$$

4) Initialize: Initialization consumes the user authorizations to transition from a contract advertisement to an active contract.

$$\begin{aligned} users(G) &= A_1, \dots, A_n & secrets &= \prod_{i=1}^n A_i [\# \, \triangleright \, \{G\}C] \\ balance(G) &= \mathbf{B} \\ & \kappa \text{ fresh} & signatures &= \prod_{i=1}^n (A_i [\{G\}C]) \\ \hline \{G\}C \mid secrets \mid signatures \mid \Gamma \xrightarrow{initialize(\kappa, \{G\}C)} \langle C, \mathbf{B} \rangle_\kappa \mid \Gamma \end{aligned} \\ S - Initialize$$

5) Abort: Abort stops the initialization of a contract, refunding all users their deposit.

$$\frac{deposits(G) = \prod_{i=1}^{n} A_{i} : !B_{i} @ \overrightarrow{x^{i}}}{\{G\}C \mid \Gamma \xrightarrow{abort(\{G\}C)} \prod_{i=1}^{n} \langle A_{i}, B_{i} \rangle_{\kappa} \mid \Gamma} S - Abort$$

APPENDIX C
COMPILING $BitML^x$ CONTRACTS

A. Introduction and Notation

1) The compiler functions: The main contract compiler functions are \mathcal{F}_C and \mathcal{F}_D for top-level contracts and guarded contracts respectively. They include a superscript \mathbb{B} indicating the target blockchain. They also take as extra input a record of settings (defined in section C-A3) that give context to the compilation.

Other auxiliary functions used during compilation are defined in the ending section.

2) Node labels and step secrets: Our implementation enforces virtually synchronous execution with a mechanism of step secrets and compensations. Every resolution of a priority choice in one blockchain will be either replicated on the other blockchains or the protocol will punish the participant responsible for the asynchronicity and compensate the rest for their loses.

To do this, we first need to tag every node in our contract's syntax tree with a unique label κ . An auxiliary function that traverses the syntax tree generating the set containing all these labels is defined in section C-E.

Then, every participant A generates for every label κ a step secret s_{κ}^{A} . The reveal of this secret serves as irrefutable of proof of A's intent to execute the subcontract labeled as κ .

The compilation of every guarded contract will require revealing a step secret from any participant but with the current node label. On section C-C1, we will use this same step secret to punish asynchronous behaviour.

3) Compiler Settings: BitMLx contracts compilation needs to keep track of many variables relating to funds, step secrets and time. However, having functions with 10 arguments would result in very poor readability. For that reason, we encapsulate everything other than the contract we are compiling in a compiler settings record:

```
\Omega ::= \{
  participants,
                                                    A list containing all the participants in the contract.
  balance,
                                                                    Logical balance at the BitMLx level.
  collateral.
                                                       How much each participant deposits as collateral.
  currentTime,
                                                                     A time counter for priority choices.
  timeElapse,
                                                                Safety time period needed for an action.
  currentLabel,
                                                                    Label of the node we are compiling.
  stepSecrets,
                            A set of all step secrets for each participant and node label in the contract.
  initSecrets,
                                                            A set of all init secrets for each participant.
}
```

We refer to an attribute in the record using dot notation. For example, $\Omega.participants$ is the participants attribute of the settings record Ω . We will use replace syntax to create updated settings records. For example, $\Omega' = \Omega[balance \mapsto b]$ means that Ω' is the record with same values as Ω , except for balance where the new value is b.

Initializing the settings record is discussed in section C-D1

4) Lists and sets by comprehension: We write lists and sets by comprehension when convenient. For example $\{f(x) \mid x \leftarrow X\}$ is the set of elements f(x) for every $x \in X$. Similarly, $[f(x_i i) \mid x_i \leftarrow \vec{x}]$ is the list whose elements are $f(x_i)$ for every element x_i of \vec{x} , in the same order.

B. Guarded Contracts

1) Withdraw: A withdraw contract is compiled as a BitML split between withdraw contract, where all participants withdraw their collateral and whatever funds they are assigned in the parameter mappings, if any.

The auxiliary function $\mathcal{F}_{Out}^{\mathbb{B}}$ that assigns the exact amount each participant withdraws, is defined below in section C-E. The result of this auxiliary function is a list of values in the cryptocurrency of \mathbb{B} .

In this guarded contract version, we also require the reveal of any step secret corresponding to this execution step.

```
\begin{split} D &= \text{withdraw } \vec{\mathbf{B}} \to \vec{A} \\ S &= \{s_{\kappa}^{A} \in \Omega.stepSecrets : \kappa = \Omega.currentLabel\} \\ \vec{\mu} &= \mathcal{F}_{Out}^{\mathbb{B}}(\vec{A}, \vec{\mathbf{B}}, \Omega) \\ \frac{\vec{C^{\mathbb{B}}} = [\text{withdraw } A_{i} \mid A_{i} \leftarrow \Omega.participants]}{\mathcal{F}_{D}^{\mathbb{B}}(D, \Omega) = \sum_{s_{\kappa}^{A} \in S} \text{reveal } s_{\kappa}^{A} \text{. split } \vec{\mu} \to \vec{C^{\mathbb{B}}} \end{split} \quad D - Withdraw \end{split}
```

2) Split: To compile a split guarded contract among subcontracts \vec{C} , we choose the appropriate funds allocation list, according to the target blockchain, compile the subcontracts and return the BitML split with both, conditioned by the reveal of any step secret.

The settings for the subcontracts compilation update the balance and collateral appropriately and add the branch index to the new label. Notice that the label used to trigger the execution (and on the other side, the punishment) does not include the branch index.

$$D = \operatorname{split} \stackrel{\rightarrow}{\mathbb{B}} \to \stackrel{\rightarrow}{C} \qquad \qquad \forall i \in 1, \dots, k : \ b_i = \underset{i}{\mathbb{B}_i} [\mathbb{B}] \\ k = |\stackrel{\rightarrow}{\mathbb{B}}| = |\stackrel{\rightarrow}{C}| \qquad \qquad \forall i \in 1, \dots, k : \ c_i = (n-2) \ b_i \\ k = |\Omega.participants| \qquad \qquad \forall i \in 1, \dots, k : \ \mu_i = b_i + n \ c_i \\ b = |\Omega.balance \qquad \qquad \forall i \in 1, \dots, k : \ \mu_i = b_i + n \ c_i \\ \forall i \in 1, \dots, k : \ \mu_i = b_i + n \ c_i \\ \forall i \in 1, \dots, k : \ \mu_i = b_i + n \ c_i \\ \forall i \in 1, \dots, k : \ \mu_i = b_i + n \ c_i \\ \text{balance} \mapsto b_i, \\ \text{collateral} \mapsto b_i, \\ \text{collateral} \mapsto c_i, \\ \text{currentLabel} \mapsto \kappa \mid i, \\ \hat{\delta} = |\Omega.timeElapse \\ S = \{s_\kappa^A \in \Omega.stepSecrets : \kappa = \Omega.currentLabel\} \\ \stackrel{\rightarrow}{C^B} = [\mathcal{F}_C^B(C_i, \Omega_i) \mid C_i \leftarrow \stackrel{\rightarrow}{C}] \\ \mathcal{F}_D^B(D, \Omega) = \sum_{s_\kappa^A \in S} \text{reveal} \ s_\kappa^A \text{. split} \ \stackrel{\rightarrow}{\mu} \to \stackrel{\rightarrow}{C^B} \\ D - Split$$

3) Authorization: An authorization is with the list of participants A is compiled to the sequential authorizations by all A_i s with the reveal of the step secret.

We are assuming that the wellFormed check prohibits D' from also being an authorization.

$$D = \vec{A} : D'$$

$$\vec{A} = A_1, \dots, A_n$$

$$\sum_{i=1}^k D_i^{'\mathbb{B}} = \mathcal{F}_D^{\mathbb{B}}(D', \Omega)$$

$$\frac{\mathcal{F}_D^{\mathbb{B}}(D, \Omega) = \sum_{i=1}^k (A_1 : \dots : A_n : D_i^{'\mathbb{B}})}{(A_n : \dots : A_n : D_i^{'\mathbb{B}})} D - Auth$$

4) Reveal: To compile a reveal statement, we compile the subcontract and wrap it with a BitML reveal conditioned by both the original secrets and the step secrets, keeping the predicate p.

$$\begin{split} D &= \text{reveal } \vec{s} \text{ if } p \text{ then } C \\ S &= \{s_{\kappa}^{\textit{A}} \in \Omega.stepSecrets : \kappa = \Omega.currentLabel\} \\ &\qquad \qquad C^{\mathbb{B}} = \mathcal{F}_{C}^{\mathbb{B}}(C,\Omega) \\ \hline &\qquad \qquad \mathcal{F}_{D}^{\mathbb{B}}(D,\ \Omega) = \sum_{s_{\alpha}^{\textit{A}} \in S} \text{reveal } \vec{s} | s_{\kappa}^{\textit{A}} \text{ if } p \text{ . } C^{\mathbb{B}} \end{split} \quad D - Reveal \end{split}$$

C. Top-level Contracts

1) Priority Choice: We want to compile a priority choice C between a guarded contract D and a contract C' with label l at time t and with time elapse $\hat{\delta}$, for the target blockchain \mathbb{B} .

We first compile D to $D^{\mathbb{B}}$, using the current label appended with an L' and increasing the time by one elapse. Similarly, we compile C' to $C'^{\mathbb{B}}$ appending R' to the label and increasing the time by two elapses.

The compilation result then, is the choice between executing $D^{\mathbb{B}}$ or, after $t + \hat{\delta}$, skipping. When skipping, we can either punish (defined further below in section C-E) or, after after $t + \hat{\delta}$, move to $C'^{\mathbb{B}}$.

$$C = D \Rightarrow C'$$

$$t = \Omega.currentTime$$

$$\hat{\delta} = \Omega.timeElapse$$

$$\kappa = \Omega.currentLabel$$

$$\Omega_D = \Omega[currentTime \mapsto t + 2\hat{\delta}, \ currentLabel \mapsto \kappa|'L']$$

$$\Omega_{C'} = \Omega[currentTime \mapsto t + 2\hat{\delta}, \ currentLabel \mapsto \kappa|'R']$$

$$D^{\mathbb{B}} = \mathcal{F}_D^{\mathbb{B}}(D, S_D)$$

$$C'^{\mathbb{B}} = \mathcal{F}_C^{\mathbb{B}}(C', S_{C'})$$

$$T^{\mathbb{B}}(C, \Omega) = D^{\mathbb{B}} + \text{after } (t + \hat{\delta}) : \tau(\mathcal{F}_{Compen}^{\mathbb{B}}(\Omega_D) + \text{after } (t + 2\hat{\delta}) : \tau(C'^{\mathbb{B}}))$$

$$C - PriorityChoice$$

The auxiliary function τ is an empty reveal statement and it works as a small hack to turn a BitML C (a sum of choices) into a D (a guarded contract). This is needed here because the after clause only takes D as inputs. The trade-off here is that doing this reveal step, is compiled as publishing a new transaction.

$$\tau(C) = reveal[] \cdot C$$

2) Withdraw: A top-level BitMLx withdraw contract is compiled in a very similar way to its guarded counterpart, only removing the requirement for revealing a step secret.

$$\begin{split} C &= \text{withdraw } \vec{v} \to \vec{A} \\ \vec{\mu} &= \mathcal{F}_{Out}^{\mathbb{B}}(\vec{A}, \ \vec{v}, \ \Omega) \\ \\ \frac{\vec{C^{\mathbb{B}}} = \left[\text{withdraw } A_i \mid A_i \leftarrow \Omega.participants \right]}{\mathcal{F}_{C}^{\mathbb{B}}(C, \Omega) = \text{split } \vec{\mu} \to \vec{C^{\mathbb{B}}}} \ C - Withdraw \end{split}$$

D. Stipulation Protocol

- 1) Initial Settings: The following function generates an initial settings record for a contract advertisement. We basically need to
 - Define the set of participants as the ones with deposits.
 - Add up the balance from the deposits on this target blockchain.
 - Determine the collaterals according to our formula.
 - Initialize the label with the empty string ϵ .
 - Compute the set of step secrets using the auxiliary functions κ_C and κ_D defined below.
 - Extract the starting time t_0 and time elapse $\hat{\delta}$ from the timing precondition.

```
\begin{split} \Omega &= \{ \\ participants &= P \\ balance &= b \\ Collateral &= (|P|-2)b \\ P &= users(G) \\ CourrentLabel &= \kappa|`L` \\ (t_0, \kappa_0) &= initSettings(G) \\ \Lambda &= nodes(C, [\kappa_0]) \\ \hline CurrentTime &= t_0 \\ currentTime &= t_0 \\ timeElapse &= \hat{\delta} \\ \hline \mathcal{F}_S^{\mathbb{B}}(\{G\}C) &= \Omega \end{split} S - Settings
```

- 2) Preconditions: When compiling preconditions, we need to:
- Pick the right deposits for the target blockchain and add the collaterals.
- Keep all regular secrets and add step secrets.

We will divide this function into two different rules, one for each target blockchain.

$$G = \underset{i=1}{\overset{n}{\parallel}} \left(A_i : ! \mathbf{B}_i @ \vec{x}\right) \qquad c = \Omega.collateral$$

$$| \underset{j=1}{\overset{m}{\parallel}} \left(A_j : \mathtt{secret} \ s_j\right) \qquad \underset{i=1}{\overset{m}{\parallel}} \left(A_i : ! (\mathbf{B}_i[\mathbb{B}] + c) @ (\vec{x}[\mathbb{B}])\right)$$

$$| \underset{j=1}{\overset{m}{\parallel}} \left(A_j : \mathtt{secret} \ s_j\right) \qquad S - Preconditions$$

$$\mathcal{F}_G^{\mathbb{B}}(G, \Omega) = G^{\mathbb{B}}$$

3) Refund: The refund contract will be executed on a cooperative abort and will just return every participant it's deposit.

$$C = \text{withdraw } \vec{\mathsf{B}} \to \vec{A} \qquad \qquad G = \prod_{i=1}^n \left(A_i : ! \, \mathsf{B}_i \, @ \, \vec{x^i}\right) \\ \vec{\mu} = \mathcal{F}_{Out}^{\mathbb{B}}(\vec{A}, \, \vec{\mathsf{B}}, \, \Omega) \qquad \qquad | \quad | \quad | \quad (A_j : \texttt{secret } s_j) \quad \vec{A} = [A_1, \, \dots, A_n] \\ \vec{C^{\mathbb{B}}} = [\texttt{withdraw } A_i \mid A_i \leftarrow \Omega.participants] \qquad | \quad | \quad | \quad (t_0, \kappa_0) \qquad \qquad S - Refund$$

$$\mathcal{F}_{Refund}(G, \Omega) = \texttt{split } \vec{\mu} \to \vec{C^{\mathbb{B}}}$$

- 4) Advertisement: This is the entry point of our compiler. To compile a contract advertisement $\{G\}C$, we need to:
- Check that the advertisement is well-formed.
- Generate the initial compilation settings.
- Compile the preconditions.
- Create a refund contract.
- Compile the contract in a priority choice with our refund contract.
- Additionally, we guard the contract with the requirement that all participants reveal their init secret. This condition means
 that all participants will explicitly agree to starting.

$$t = \Omega.currentTime$$

$$\hat{\delta} = \Omega.timeElapse$$

$$\kappa = \Omega.currentLabel$$

$$\Omega_D = \Omega[currentTime \mapsto t + 2\hat{\delta}, \ currentLabel \mapsto \kappa|'L'|$$

$$\Omega_{C'} = \Omega[currentTime \mapsto t + 2\hat{\delta}, \ currentLabel \mapsto \kappa|'K'|$$

$$P = users(G)$$

$$(t_0, \kappa_0) = initSettings(G)$$

$$IS = \bigcup_{A \in P} \{is_{\kappa_0}^A\}$$

$$D^B = \mathcal{F}_D^B((\text{reveal } IS \text{ then } C), \Omega_D)$$

$$C'^B = \mathcal{F}_C^B(\mathcal{F}_{Refund}(G), \Omega_{C'})$$

$$\mathcal{F}_{Stip}^B(\{G\}C, \Omega) = D^B + \text{after } (t + \hat{\delta}) : \tau(\mathcal{F}_{Compen}^B(\Omega_D) + \text{after } (t + 2\hat{\delta}) : C'^B)$$

$$S - Stipulation$$

$$wellFormed(\{G\}C)$$

$$\forall B \in chains(G) : \Omega^B = \mathcal{F}_S^B(\{G\}C_x)$$

$$\forall B \in chains(G) : G^B = \mathcal{F}_G^B(G, \Omega^B)$$

$$\forall B \in chains(G) : C^B = \mathcal{F}_{Stip}^B(\{G\}C, \Omega^B)$$

$$\overline{\{G\}C} = [\{G^B\}C^B : \forall B \in chains(G)]$$

$$S = \mathcal{F}_{PS}(G)$$

$$\mathcal{F}_{Adv}(\{G\}C) = \{S\}[\overline{\{G\}C'\}}]$$

$$S - Advertise$$

E. Auxiliary Functions

1) Step secret generation:

$$nodes(D \Rightarrow C, \kappa) = nodes_D(D, \kappa|'L') \cup nodes(C, \kappa|'R')$$

 $nodes(\text{withdraw } \vec{B} \rightarrow \vec{A}, \kappa) = \{\}$

$$nodes_D(ext{split }\vec{\mathbf{B}} o \vec{C}, \kappa) = \{\kappa\} \ \cup \ \bigcup_{C_i \in \vec{C}} nodes(C_i, \kappa|i)$$

$$nodes_D(ext{reveal } \vec{s} \text{ if } p \text{ then } C, \kappa) = \{\kappa\} \ \cup \ nodes(C, \kappa)$$

$$nodes_D(\vec{A}: D, \kappa) = nodes_D(D, \kappa)$$

$$nodes_D(\text{withdraw } \vec{\mathbf{B}} \to \vec{A}, \kappa) = \{\kappa\}$$

2) Compensate: The compensation contract is a sum of reveals where for each step secret at the current label, we give every participant, except the owner of the revealed step secret, a balance b worth of the cryptocurrency on \mathbb{B} .

$$\begin{split} A_1, & \ldots, & A_n = \Omega.participants \\ & b = \Omega.balance \\ & \kappa = \Omega.currentLabel \\ & \forall i \in [1, \ldots, n]: \ s_{\kappa}^{A_i} \in \Omega.stepSecrets \\ & \forall i \in [1, \ldots, n]: \ \vec{v^i} = (b)^{n-1} \\ & \frac{\forall i \in [1, \ldots, n]: \ \vec{C^i} = [\text{withdraw } A_j \mid j \leftarrow [1, \ \ldots, \ n] \backslash \{i\}]}{\mathcal{F}_{Compen}^{\mathbf{B}}(\Omega) = \sum_{i=1}^n \text{reveal } s_{\kappa}^{A_i} \ . \ (\text{split } \vec{v^i} \rightarrow \vec{C^i}) \end{split}$$
 Compensate

3) Withdraw outputs: The following functions compute how many bitcoins and how many dogecoins should withdraw each participant, by adding their collateral, and whatever was stateted in the BitMLx withdraw statement.

$$\vec{A} = A_1, \dots, A_n$$

$$\vec{B} = B_1, \dots, B_n$$

$$c = \Omega.collateral$$

$$P = \Omega.participants$$

$$\forall A \in P: \ v_A = \begin{cases} B_i[\mathbb{B}] + c & \text{if } \exists i: A = A_i \\ c & \text{otherwise} \end{cases}$$

$$\overrightarrow{\mathcal{F}_{Out}^{\mathbb{B}}(\vec{A}, \ \vec{B}, \ \Omega)} = [v_A \mid A \leftarrow P]$$

$$\overrightarrow{APPENDIX D}$$

$$\overrightarrow{APPENDIX D}$$

$$\overrightarrow{APPENDIX D}$$

$$\overrightarrow{APPENDIX D}$$

$$\overrightarrow{APPENDIX D}$$

A. Configurations

Intermediate semantics configurations look very similar to $BitML^x$ configurations, but deposits and contracts reside now on a single blockchain, indicated by a superscript. They also have a state record π , with a lot more context on the contract, instead of just the balance. One of them, is the "contract status", which models an internal state machine for the contracts. The status can be:

- A Choice contract is in it's neutral state, waiting to de between the left or right move.
- A Left contract has executed the reveal statement corresponding to the step secret, but didn't execute the inner statement
 yet.
- An **Assigned**(A) contract has finished executing and can be withdrawn by A.
- A **Right** contract has executed the τ step of disabling the left move and is waiting for possible compensations.
- A Slashed(A) contract has executed the reveal statement corresponding to compensating for a late step secret reveal from user A, but has not executed the inner split yet.
- A CompensatedFrom(A) contract has executed the split statement corresponding to compensating for a late step secret reveal from user A.
- A Stip-Choice contract is going through the stipulation protocol. It is just published and needs to either initialize or abort.
- A Stip-Right contract is going through the stipulation protocol. It has taken the sright τ step and is waiting for possible compensations before refunding everyone's deposit.
- A **Stip-Slashed**(A) contract is going through the stipulation protocol. It has executed the reveal statement corresponding to compensating for a late initial step secret reveal from user A, but has not executed the inner split yet.
- A **Stip-Compensation**(A) contract is going through the stipulation protocol. It has executed the split statement corresponding to compensating for a late initial step secret reveal from user A.

• A **Stip-Refunded**(A) contract is going through the stipulation protocol. It has executed the split statement corresponding to refunding their deposits and is now assigned to user A.

Finally, there are new secret objects that explicitly differentiate step secrets and init secrets from logical secrets (secrets that are part of the $BitML^x$ contract logic).

$$\begin{split} \Gamma^{||} &\coloneqq \{G\}^{\mathbb{B}} C \\ &\mid \langle C, \pi \rangle_{\kappa, x}^{\mathbb{B}} \\ &\mid \langle A, v \rangle_{x}^{\mathbb{B}} \\ &\mid A : [\phi] \\ &\mid \{A : s \# N\} \\ &\mid A : s \# N \\ &\mid \{A : s_{\kappa}^{A}\} \\ &\mid A : s_{\kappa}^{A} \} \\ &\mid A : i s_{\kappa}^{A} \} \\ &\mid A : i s_{\kappa}^{A} \\ &\mid \Gamma^{||} \|\hat{\delta}^{||} \\ \Gamma^{||} \mid t \end{split}$$

Active contracts need to store a lot of additional information, so we hide all of this into a state record π .

```
\begin{split} \pi &:= \{\\ balance : v\\ status : s\\ time : t\\ participants : P\\ deposits : D\\ \} \end{split}
```

B. Secrets

The |-RevealSecret| rule says that if A really has the value for a secret they committed, then they can always reveal it. Remember that dishonest users can commit to any secret, even if they don't actually know the value, which is represented by the \bot length.

$$\frac{N \neq \bot}{\{ {\color{red} A}: a\# N \} \mid \Gamma^{||} \xrightarrow{{\color{blue} A}: a} {\color{blue} A}: a\# N \mid \Gamma^{||}} \mid| - RevealSecret}$$

C. Priority choice internals

These rules handle the tracking and use of step secrets to transition among the different internal states we use to implement priority choices.

1) Authorize a left: Moving a contract to an intermediate left state requires a revealed step secret. Any committed step secret can be revealed at any time. Honest users will only do so when they want to execute a left side choice, but they need to be able to parse and keep track of malicious users revealing theirs at any time.

$$\frac{}{\{\textbf{\textit{A}}:\ s_{\kappa}^{\textbf{\textit{A}}}\}\mid \Gamma^{||}\xrightarrow{\textbf{\textit{A}}:\ s_{\kappa}^{\textbf{\textit{A}}}}\textbf{\textit{A}}:\ s_{\kappa}^{\textbf{\textit{A}}}\mid \Gamma^{||}}\mid |-StepSecretRev$$

2) Introduction of left: This step corresponds to executing the reveal statement guarded by the step secret and transitioning to the actual D contract, when D is not a reveal itself. The left modifier will then be eliminated by the actual execution of D to the rules on the next section. The step secret is not consumed in the transition and it can be used in the other chain for either replication or compensation.

We additionally require any needed signatures for auth guards.

$$\begin{split} D \neq \text{reveal } \vec{s} \text{ if } p \text{ then } C' \\ D \neq \vec{A'} : D' \\ \pi = [status = \textbf{Choice}, \ \dots] \\ sigs = & \parallel A_i[(\kappa, \mathbb{B}, x) \rhd D] \\ A_i \in \vec{A} \\ \pi' = \pi[status \mapsto \textbf{Left}] \\ x' \text{ fresh} \\ \hline \langle \vec{A} : D \Rightarrow C, \pi \rangle_{\kappa, x}^{\mathbb{B}} \mid A : \ s_{\kappa}^{A} \mid sigs \mid \Gamma^{||} \xrightarrow{ileft(\kappa, \mathbb{B}, x)} \langle D \Rightarrow C, \pi \rangle_{\kappa, x'}^{\mathbb{B}} \mid A : \ s_{\kappa}^{A} \mid \Gamma^{||} \end{split} \\ | - ILeft$$

The reveal statement needs special treatment. The reveal of $BitML^x$ secrets is merged with that of step secrets and there is no intermediate state before transitioning to the follow-up contract.

$$D = \text{reveal } \vec{s} \text{ if } p \text{ then } C'$$

$$\pi = [time = t, status = \textbf{Choice}, \dots]$$

$$sigs = \prod_{\substack{A_i \in \vec{A} \\ A_i \in \vec{A}}} A_i[(\kappa, \mathbb{B}, x) \rhd D]$$

$$secrets = \prod_{\substack{S_j \in \vec{s} \\ s_j \in \vec{s}}} A_j : s_j \# N_j$$

$$[p]_{secrets} = true$$

$$\pi' = \pi[time \mapsto t + 2\hat{\delta}]$$

$$\kappa' = \kappa | L$$

$$| A : s_{\kappa}^A \mid secrets \mid sigs \mid \Gamma^{||} \xrightarrow{reveal(\kappa, \mathbb{B}, x)} \langle C', \pi' \rangle_{\kappa', x'}^{\mathbb{B}} \mid A : s_{\kappa}^A \mid secrets \mid \Gamma^{||}$$

$$| - Reveal \mid A : s_{\kappa}^A \mid secrets \mid sigs \mid \Gamma^{||} \xrightarrow{reveal(\kappa, \mathbb{B}, x)} \langle C', \pi' \rangle_{\kappa', x'}^{\mathbb{B}} \mid A : s_{\kappa}^A \mid secrets \mid \Gamma^{||}$$

The signatures can be added with ||-AuthControl| rule.

$$\pi = [status = \textbf{Choice}, \dots]$$

$$\frac{A_i \in \vec{A}}{\langle \vec{A} : D \Rightarrow C, \pi \rangle_{\kappa, x}^{\mathbb{B}} \mid \Gamma^{||} \xrightarrow{A:(\kappa, \mathbb{B}, x)} \langle \vec{A} : D \Rightarrow C, \pi \rangle_{\kappa, x}^{\mathbb{B}} \mid A_i[(\kappa, \mathbb{B}, x) \triangleright D] \mid \Gamma^{||}} \mid| - AuthControl$$

3) Introduction of right: ISkip moves the contract to an intermediate state where we can either move right or compensate. This corresponds in the BitML level to executing the τ (empty reveal) statement to discard D.

$$\begin{split} \pi &= [time = t, status = \textbf{Choice}, \ \dots] \\ \frac{\pi' = \pi[time \mapsto t + \hat{\delta}, \ status \mapsto \textbf{Right}]}{\langle D \Leftrightarrow C, \pi \rangle_{\kappa, x}^{\mathbb{B}} \mid \Gamma^{||} \xrightarrow{from \ t: \ skip(\kappa, \mathbb{B}, x)} \langle D \Leftrightarrow C, \pi' \rangle_{\kappa, x'}^{\mathbb{B}} \mid \Gamma^{||}} \mid | - IRight \end{split}$$

4) Elimination of right: A skip contract can finish skipping and move to the right side of the priority choice after some time condition is met. The label gets updated accordingly.

$$\begin{split} \pi &= [time = t, status = \textbf{Right}, \ \dots] \\ \pi' &= \pi[time \mapsto t + \hat{\delta}, \ status \mapsto \textbf{Choice}] \\ x' \text{ fresh} \\ \frac{\kappa' = \kappa |R|}{\langle D \Leftrightarrow C', \pi \rangle_{\kappa, x}^{\mathbb{B}} \mid \Gamma^{||} \xrightarrow{from \ t: \ right(\kappa, \mathbb{B}, x)} \langle C', \pi' \rangle_{\kappa', x'}^{\mathbb{B}} \mid \Gamma^{||}} \mid| - ERight \end{split}$$

5) Introduction of Compensation: A skipping contract can also transition to a compensation when there's a step secret for the corresponding left choice in the context. Again, the step secret is not consumed because it might be needed for the other blockchain.

Notice that we transition to an intermediate state and not directly to the withdraw states, because this is implemented as a reveal step.

$$\begin{split} \pi &= [status = \textbf{Right}, \quad \ldots] \\ \pi' &= \pi [status \mapsto \textbf{Slashed}(\textbf{A})] \\ &\qquad \qquad x' \text{ fresh} \\ \hline \langle D \Leftrightarrow C, \pi \rangle_{\kappa, x}^{\mathbb{B}} \mid \textbf{A}: \ s_{\kappa}^{\textbf{A}} \mid \Gamma^{||} \xrightarrow{slash(\kappa, \mathbb{B}, x, \textbf{A})} \langle D \Leftrightarrow C, \pi' \rangle_{\kappa, x'}^{\mathbb{B}} \mid \textbf{A}: \ s_{\kappa}^{\textbf{A}} \mid \Gamma^{||} \end{split} \\ || - ICompensation$$

6) Elimination of Compensation: A compensation is just a withdraw with a special name that will help us later on. Because we move to withdrawals, we add the collaterals.

$$\begin{split} \pi &= [status = \mathbf{Slashed}(A), \ldots] \\ \pi' &= \pi [status \mapsto \mathbf{CompensatedFrom}(A)] \\ \frac{\vec{x} \text{ fresh}}{\langle C, \pi \rangle_{\kappa, x}^{\mathbb{B}} \mid \Gamma^{||} \xrightarrow{compensate(\kappa, \mathbb{B}, x, A)} \langle C, \pi' \rangle_{\kappa, \vec{x}}^{\mathbb{B}} \mid \Gamma^{||}} \mid| - ECompensation \end{split}$$

D. Eliminations of left

After the left state has been introduced, guarded contracts basically transition just like they would in $BitML^x$. Most importantly, they update the κ when needed.

1) Guarded Withdraw: A withdraw contract in intermediate left state transitions to assigned funds for each specified participant on the corresponding blockchain.

$$\begin{split} D &= \texttt{withdraw} \stackrel{\vec{\mathbf{B}}}{\rightarrow} \stackrel{\vec{A}}{A} \\ \pi &= [status = \mathbf{Left}, \ \dots] \\ \vec{\mathbf{B}} &= \mathbf{B}_1, \dots, \mathbf{B}_n \\ \vec{A} &= A_1, \dots, A_n \\ \forall i \in 1, \dots, n : \pi_i = \pi[balance \mapsto \mathbf{B}_i[\mathbb{B}], \ status \mapsto \mathbf{Assigned}(A_i)] \\ \forall i \in 1 \dots n : \ \kappa_i = \kappa | L_i \\ \hline \frac{x_1, \dots, x_n \ \text{fresh}}{\langle D \Leftrightarrow C, \pi \rangle_{\kappa, x}^{\mathbb{B}} \mid \Gamma^{||} \stackrel{dwithdraw(\kappa, \mathbb{B}, x)}{\rightarrow} \prod_{i=1}^n \langle D \Leftrightarrow C, \pi' \rangle_{\kappa_i, x_i}^{\mathbb{B}} \mid \Gamma^{||}} \mid| - GuardedWithdraw \end{split}$$

2) Split: A split left contract transitions to the active subcontracts, each with their corresponding balance and label.

E. Top-level Withdraw

A right-most withdraw needs no step secret and can always be executed.

$$C = \text{withdraw } \overrightarrow{\mathbb{B}} \to \overrightarrow{A}$$

$$\overrightarrow{A} = A_1, \dots, A_n$$

$$\overrightarrow{\mathbb{B}} = \mathbb{B}_1, \dots, \mathbb{B}_n$$

$$\pi = [status = \textbf{Choice}, \dots]$$

$$\forall i \in 1, \dots, n : \pi_i = \pi[balance \mapsto \mathbb{B}_i[\mathbb{B}], \ status \mapsto \textbf{Assigned}(A_i)]$$

$$\forall i \in 1 \dots n : \ \kappa_i = \kappa | L_i$$

$$\underbrace{x_1, \dots, x_n \ \text{fresh}}_{i=1} \langle C, \pi \rangle_{\kappa_i, x_i}^{\mathbb{B}} \mid \Gamma^{||} \xrightarrow{cwithdraw(\kappa, \mathbb{B}, x)} \prod_{i=1}^n \langle C, \pi \rangle_{\kappa_i, x_i}^{\mathbb{B}} \mid \Gamma^{||}$$

$$|| - Withdraw$$

F. Stipulation Protocol

We advertise all contracts simultaneously, corresponding to advertising a batch. We require users to have enough funds to cover for both the balance and their collateral.

Then of course, we have regular advertisement conditions such as using fresh secrets, having at least one honest user and for the advertisement to be well-formed.

In the commit phase, users commit to their logical secrets, their step secrets and their init secrets. These are represented as global secrets in the batch and that's why this is a single action. The *nodes* function is defined in the compiler theory document.

We don't require freshness conditions on the step and init secrets because we are already using a fresh κ .

Init authorizations are by contract, because each contract requires it's own signatures.

$$\begin{split} \forall A_i \in users(G). \ A_i : [\# \rhd \{G\}^{\mathbb{B}}C] \in \Gamma^{||} \\ A : !B @ \vec{x} \in G \\ \hline x \in \vec{x} \\ \hline \{G\}^{\mathbb{B}}C \mid \Gamma^{||} \xrightarrow{authInit(A,\{G\}^{\mathbb{B}}C)} \{G\}^{\mathbb{B}}C \mid A : [x \rhd \{G\}^{\mathbb{B}}C] \mid \Gamma^{||} \end{split}$$

An authorized contract can be published on it's blockchain, consuming the deposits and authorizations on it. The result is a starting contract, that will go through the synchronization protocol.

$$P = users(G) \qquad \pi = [$$

$$v = balance_{\mathbb{B}}(G) \qquad status = \mathbf{Stip\text{-}Choice},$$

$$D = deposits(G) \qquad balance = v,$$

$$(t_0, \kappa_0) = initSettings(G) \qquad participants = P,$$

$$\Delta = \underset{A_i \in P}{\parallel} (\langle A_i, v_i + c \rangle_{x_i}^{\mathbb{B}} \mid A_i : [\# \triangleright \{G\}^{\mathbb{B}}C] \mid A_i : [x_i \triangleright \{G\}^{\mathbb{B}}C]) \qquad time = t_0,$$

$$deposits = D,$$

$$x \text{ fresh} \qquad |$$

$$\{G\}^{\mathbb{B}}C \mid \Delta \mid \Gamma^{||} \xrightarrow{publish(\{G\}^{\mathbb{B}}C)} \langle C, \pi \rangle_{\kappa_0, x}^{\mathbb{B}} \mid \Gamma^{||} \qquad || - Publish$$

Alternatively, a user can double-spend the funds reserved as their deposit in the advertisement, rendering it useless. We are interested in distinguishing this case, because we can use it to emit a specific transition label.

$$\frac{A: ! \ \mathbb{B} \ @ \vec{x} \in G}{x \in \vec{x}} \\ \frac{x \in \vec{x}}{\langle A, v \rangle_x^{\mathbb{B}} \mid \{G\}^{\mathbb{B}}C \mid \Gamma^{||} \xrightarrow{doubleSpend(\{G\}^{\mathbb{B}}C, A, x)} \{G\}^{\mathbb{B}}C \mid \Gamma^{||}} \mid | -DoubleSpendDeposit$$

Assuming all contract have been published, then honest users can start revealing their stipulation secrets. The rule is very similar to revealing a step secret.

$$\frac{}{\{\boldsymbol{A}:is^{\boldsymbol{A}}_{\kappa_0}\}\mid \Gamma^{||} \xrightarrow{\boldsymbol{A}:is^{\boldsymbol{A}}_{\kappa_0}} \boldsymbol{A}:is^{\boldsymbol{A}}_{\kappa_0}\mid \Gamma^{||}} \mid| -InitSecretRev$$

If everyone revealed their stipulation secrets and someone also revealed a corresponding step secret, we can initialize the contract, resulting in an active contract, ready to start running.

$$\begin{split} \pi &= [\\ status &= \textbf{Stip-Choice}, \quad \pi' = \pi[\\ time &= t_0, \qquad status \mapsto \textbf{Choice}, \\ participants &= P, \qquad time \mapsto t_0 + 2\hat{\delta}, \\ & \cdots \\ \end{bmatrix} \\ \hat{\delta} &= A: \ s_{\kappa_0}^{\pmb{A}} \mid \underset{\pmb{A_i} \in P}{\parallel} A_{\pmb{i}}: is_{\kappa_0}^{\pmb{A_i}} \\ \hline \langle C, \pi \rangle_{\kappa_0, x}^{\pmb{B}} \mid \hat{\delta} \mid \Gamma^{||} \xrightarrow{init(\kappa_0, \pmb{B}, x)} \langle C, \pi' \rangle_{[\kappa_0], x'}^{\pmb{B}} \mid \Gamma^{||} \end{split} \\ \mid -Init \end{split}$$

After the timeout, we can abort the contract, transitioning it to an aborted contract.

$$\begin{split} \pi &= [time = t, \; status = \textbf{Stip-Choice}, \; \dots] \\ \pi' &= \pi[time \mapsto t + \hat{\delta}, \; status \mapsto \textbf{Stip-Right}] \\ &\frac{x' \; \text{fresh}}{\langle C, \pi \rangle_{\kappa_0, x}^{\mathbb{B}} \mid \Gamma^{||} \; \frac{from \; t: \; sright(\kappa_0, \mathbb{B})}{\langle C, \pi \rangle_{\kappa_0, x'}^{\mathbb{B}} \mid \Gamma^{||}} \mid | - SSkip \end{split}$$

After the timeout, an aborted contract can be refunded to all participants

$$\begin{split} \pi = [status = \textbf{Stip-Right}, \ time = t, \ participants = P, \ deposits = D, \ \dots] \\ \forall A_i \in P : (A_i, v_i \mathbb{B}) \in D \\ \forall A_i \in P : \pi_i = \pi [balance \mapsto v_i, \ status \mapsto \textbf{Stip-Refunded}(A_i)] \\ \hline & \forall A_i \in P : x_i \ \text{fresh} \\ \hline & \langle C, \pi' \rangle_{\kappa_0, x}^{\mathbb{B}} \mid \Gamma^{||} \xrightarrow{from \ t: \ abort(\kappa_0, \mathbb{B}, x)} & \parallel_{A_i \in P} \langle C, \pi_i \rangle_{\kappa_i, x_i}^{\mathbb{B}} \mid \Gamma^{||} \end{split}$$

If we have a step secret, the aborted contract can transition to a compensation for all the participants, except the owner of the step secret.

$$\pi = [status = \textbf{Stip-Right}, \ \dots]$$

$$\pi' = \pi[status \mapsto \textbf{Stip-Compensation}(A)]$$

$$x' \text{ fresh}$$

$$\langle C, \pi \rangle_{\kappa_0, x}^{\mathbb{B}} \mid A: \ s_{\kappa_0}^{A} \mid \Gamma^{||} \xrightarrow{sslash(\kappa_0, \mathbb{B}, A, x)} \langle C, \pi' \rangle_{\kappa_0, x'}^{\mathbb{B}} \mid A: \ s_{\kappa_0}^{A} \mid \Gamma^{||}$$

$$|| - StipCompensation$$

$$\pi = [status = \textbf{Stip-Slashed}(A), \ \dots]$$

$$\pi' = \pi[status \mapsto \textbf{Stip-Compensation}(A)]$$

$$\vec{x} \text{ fresh}$$

$$|| - EStipCompensation$$

$$\langle C, \pi \rangle_{\kappa_0, x}^{\mathbb{B}} \mid \Gamma^{||} \xrightarrow{scompensate(\kappa_0, \mathbb{B}, A, x)} \langle C, \pi' \rangle_{\kappa_0, \vec{x}}^{\mathbb{B}} \mid \Gamma^{||}$$

G. Timed Semantics

Timed semantics are like a higher level to base semantics described so far. They describe how to transition in cases where labels have time conditions like before and from. Note that our from is called after in BitML.

1) Action: Actions without time conditions can and enabled on an untimed context, can be performed with any time on a timed context.

$$\frac{\Gamma_0^{||} \xrightarrow{\alpha} \Gamma^{||}}{\Gamma_0^{||} \mid t \xrightarrow{\alpha} \Gamma^{||} \mid t} T - Action$$

2) Delay: Time can always be advanced.

$$\frac{\hat{\delta} > 0}{\Gamma^{||} \mid t \xrightarrow{\hat{\delta}} \Gamma^{||} \mid t + \hat{\delta}} T - Delay$$

3) From: This rule eliminates from conditionals on transitions. Notice that we call it from and not after as in BitML, because the range is inclusive.

$$\begin{split} & \Gamma_0^{||} \xrightarrow{from \ t': \ \alpha} \Gamma^{||} \\ & \frac{t \geq t'}{\Gamma_0^{||} \ | \ t \xrightarrow{\alpha} \Gamma^{||} \ | \ t} \ T - From \end{split}$$

APPENDIX E COMPILING $BitML^x$ STRATEGIES

Given a $BitML^x$ user strategy Σ_A^x we define by the following rules a function $\Sigma_A^{||} = \mathcal{S}(\Sigma_A^x)$ that produces the compiled $BitML^{||}$ strategy.

A. Active Contracts

1) Left Move:

$$\begin{split} & \underbrace{A \in \mathcal{R}^{ss}_{\kappa}(R^{||})}_{R^{||} \xrightarrow{reveal(\kappa, \mathbb{B})} & \xrightarrow{reveal(\kappa, \mathbb{B}) \in \hat{\Sigma}^{||}_{A}(R^{||})} S - Reveal \end{split}$$

$$\frac{A \in \mathcal{R}^{ss}_{\kappa}(R^{||})}{R^{||} \xrightarrow{ileft(\kappa,\mathbb{B})}} \xrightarrow{S - ILeft}$$
$$ileft(\kappa,\mathbb{B}) \in \hat{\Sigma}^{||}_{A}(R^{||})$$

$$\begin{array}{c} R^{\mid\mid} \xrightarrow{\alpha(\kappa,\mathbb{B})} \\ \frac{\alpha \in \{dwithdraw, split\}}{\alpha(\kappa,\mathbb{B}) \in \hat{\Sigma}_{A}^{\mid\mid}(R^{\mid\mid})} \ S - ELeft \end{array}$$

2) Right and Skip:

$$\frac{R^{\mid\mid} \xrightarrow{right(\kappa,\mathbb{B})}}{right(\kappa,\mathbb{B}) \in \hat{\Sigma}_{A}^{\mid\mid}(R^{\mid\mid})} \ S - IRight$$

$$\frac{R^{||} \xrightarrow{skip(\kappa,\mathbb{B})}}{skip(\kappa,\mathbb{B}) \in \hat{\Sigma}_{A}^{||}(R^{||})} S - ERight$$

3) Compensation:

$$\frac{R^{\mid\mid} \xrightarrow{slash(\kappa,\mathbb{B},B)}}{slash(\kappa,\mathbb{B},x,B) \in \hat{\Sigma}_{A}^{\mid\mid}(R^{\mid\mid})} S - Slash$$

$$\frac{R^{||} \xrightarrow{compensate(\kappa,\mathbb{B},x,B)}}{compensate(\kappa,\mathbb{B},B) \in \hat{\Sigma}_{A}^{||}(R^{||})} \ S-Compensate$$

4) Authorizations:

$$\begin{split} R^x \sim_{A} R^{||} \\ (A:\kappa) \in \Sigma_{A}^x(R^x) \\ & \xrightarrow{A:(\kappa,\mathbb{B})} \xrightarrow{A:(\kappa,\mathbb{B})} S - AuthControlStart \end{split}$$

$$\frac{A \in \mathcal{A}_{\kappa}(R^{||})}{R^{||} \xrightarrow{A:(\kappa,\mathbb{B})}} \xrightarrow{A:(\kappa,\mathbb{B}) \in \hat{\Sigma}_{A}^{||}(R^{||})} S - AuthControlFinish$$

$$R^{x} \sim_{A} R^{||}$$

$$\frac{(A:a) \in \Sigma_{A}^{x}(R^{x})}{(A:a) \in \hat{\Sigma}_{A}^{||}(R^{||})} S - RevealSecret$$

B. Pre-Stipulation

$$\begin{split} R^x \sim_{\pmb{A}} R^{||} \\ advertise(\{G\}C) \in \Sigma^x_{\pmb{A}}(R^x) \\ &\xrightarrow{R^{||}} \xrightarrow{advertise(\{G\}C)} S - Adv - Advertise \\ advertise(\{G\}C) \in \hat{\Sigma}^{||}_{\pmb{A}}(R^{||}) \end{split}$$

$$\begin{split} R^x \sim_{\pmb{A}} R^{||} \\ commit(\pmb{A}, \{G\}C) \in \Sigma_{\pmb{A}}^x(R^x) \\ & \xrightarrow{R^{||}} \xrightarrow{commit(\{G\}C)} \\ \hline commit(\pmb{A}, \{G\}C) \in \hat{\Sigma}_{\pmb{A}}^{||}(R^{||})} \ S - Adv - AuthCommit \end{split}$$

$$R^{x} \sim_{A} R^{||}$$

$$(A: \{G\}C) \in \Sigma_{A}^{x}(R^{x})$$

$$R^{||} \xrightarrow{A: \{G\}^{\mathbb{B}C}}$$

$$(t_{0}, \kappa_{0}) = initSettings(G)$$

$$t < t_{0}$$

$$(A: \{G\}^{\mathbb{B}C}) \in \hat{\Sigma}_{A}^{||}(R^{||})$$

$$S - Adv - AuthInitStart$$

$$\frac{A \in \mathcal{A}_{\kappa_0}(R^{||})}{R^{||} \xrightarrow{A:\{G\}^{\mathbb{B}}C}} \xrightarrow{} S - AuthInitFinish$$

$$\frac{(A:\{G\}^{\mathbb{B}}C) \in \hat{\Sigma}_A^{||}(R^{||})}{A} = AuthInitFinish$$

C. x-Stipulation

$$\begin{split} R^x \sim_A R^{||} \\ (t_0, \kappa_0) &= initSettings(G) \\ init(\kappa_0) \in \Sigma_A^x(R^x) \\ R^{||} &\xrightarrow{publish(\{G\}^{\mathbb{B}}C)} \\ & t < t_0 \\ \hline publish(\{G\}^{\mathbb{B}}C) \in \hat{\Sigma}_A^{||}(R^{||}) \end{split} S - Adv - Publish \end{split}$$

$$\begin{split} R^x \sim_{\pmb{A}} R^{||} \\ (t_0, \kappa_0) &= initSettings(G) \\ not \ R^{||} &\xrightarrow{publish(\{G\}^{\mathbb{B}}C)} \\ &\xrightarrow{t \geq t_0} \\ &\xrightarrow{doubleSpend(\{G\}^{\mathbb{B}}C, \pmb{A}) \in \hat{\Sigma}^{||}_{\pmb{A}}(R^{||})} S - Adv - DoubleSpend \end{split}$$

$$R^{x} \sim_{A} R^{||}|t$$

$$(t_{0}, \kappa_{0}) = initSettings(G)$$

$$init(\kappa_{0}) \in \Sigma_{A}^{x}(R^{x})$$

$$\forall \mathbb{B} \in \mathcal{B} : \exists \langle C, \pi \rangle_{\kappa_{0}}^{s} \in \Gamma_{R^{||}}$$

$$A \notin \mathcal{R}_{\kappa_{0}}^{is}(R^{||})$$

$$t < t_{0}$$

$$is_{\kappa_{0}}^{A} \in \hat{\Sigma}_{A}^{||}(R^{||})$$

$$S - Stip - InitSecret$$

$$R^{x} \sim_{A} R^{||}|t$$

$$(t_{0}, \kappa_{0}) = initSettings(G)$$

$$init(\kappa_{0}) \in \Sigma_{A}^{x}(R^{x})$$

$$\mathcal{R}_{\kappa_{0}}^{is}(R^{||}) = users(\kappa_{0})$$

$$A \notin \mathcal{R}_{\kappa_{0}}^{ss}(R^{||})$$

$$t < t_{0}$$

$$S_{\kappa_{0}}^{A} \in \hat{\Sigma}_{A}^{||}(R^{||})$$

$$S - Stip - StepSecret$$

$$R^{x} \sim_{A} R^{||}$$

$$A \in \mathcal{R}_{\kappa_{0}}^{ss}(R^{||})$$

$$R^{||} \frac{init(\kappa_{0}, \mathbb{B})}{sinit(\kappa_{0}, \mathbb{B}) \in \hat{\Sigma}_{A}^{||}(R^{||})}$$

$$S - Stip - Initialize$$

$$R^{x} \sim_{A} R^{||}$$

$$abort(\kappa_{0}) \in \Sigma_{A}^{x}(R^{x})$$

$$R^{||} \frac{doubleSpend(\kappa_{0}, \mathbb{B}, A)}{sinit(\kappa_{0}, \mathbb{B}) \in \hat{\Sigma}_{A}^{||}(R^{||})}$$

$$S - Stip - Abort$$

$$\frac{R^{||} \frac{sright(\kappa_{0}, \mathbb{B})}{sright(\kappa_{0}, \mathbb{B})} \in \hat{\Sigma}_{A}^{||}(R^{||})}$$

$$S - Stip - Abort$$

$$\frac{R^{||} \frac{sright(\kappa_{0}, \mathbb{B})}{abort(\kappa_{0}, \mathbb{B})} \in \hat{\Sigma}_{A}^{||}(R^{||})$$

$$S - Stip - Abort$$

$$\frac{R^{||} \frac{abort(\kappa_{0}, \mathbb{B})}{abort(\kappa_{0}, \mathbb{B})} \in \hat{\Sigma}_{A}^{||}(R^{||})$$

$$S - Stip - Abort$$

$$R^{||} \frac{scompensate(\kappa, \mathbb{B}, x, A')}{abort(\kappa_{0}, \mathbb{B})} \in \hat{\Sigma}_{A}^{||}(R^{||})$$

$$S - Stip - Abort$$

$$\frac{R^{||} \frac{abort(\kappa_{0}, \mathbb{B})}{abort(\kappa_{0}, \mathbb{B})} \in \hat{\Sigma}_{A}^{||}(R^{||})}$$

$$S - Stip - Abort$$

$$S - Stip - Abort$$

$$R^{||} \frac{abort(\kappa_{0}, \mathbb{B})}{abort(\kappa_{0}, \mathbb{B})} \in \hat{\Sigma}_{A}^{||}(R^{||})$$

$$S - Stip - Abort$$

$$S - Sti$$

D. Timed Strategy

$$\frac{\hat{\Sigma}_{A}^{||}(R^{||}) \neq \emptyset}{\Sigma_{A}^{||}(R^{||}) = \hat{\Sigma}_{A}^{||}(R^{||})} S - Action$$

$$\mathcal{T}(R^{||}|t) = \min\{\hat{\delta} : \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma^{x}(R^{||}) \wedge t + \hat{\delta} = \pi.timeout\}$$

$$\begin{split} \hat{\Sigma}_{A}^{\mid\mid}(R^{\mid\mid}) &= \emptyset \\ \frac{\hat{\delta} &= \mathcal{T}(R^{\mid\mid})}{\Sigma_{A}^{\mid\mid}(R^{\mid\mid}) &= \{\hat{\delta}\} \end{split} S - Wait \end{split}$$

APPENDIX F FRONTIERS

A. Frontier Definitions

We will introduce the concept of a *contract frontier* in a run. Intuitively, a frontier is a set of node identifiers that covers all contract executions in the run.

More formally we define for $BitML^x$ runs R^x :

Definition 2 (BitML^x frontiers). Let R^x be a BitML^x run. Then a set F of identifiers is a frontier of R^x (written frontier (F, R^x)) if the following conditions hold:

- 1) $\forall \kappa \in F$. $\kappa \in \mathbb{R}^x$ i.e. κ is in some configuration of \mathbb{R}^x
- 2) $\forall \kappa' \in R^x$. $\exists \kappa \in F$. $\kappa \in anc(\kappa') \lor \kappa \in desc(\kappa')$
- 3) $\forall \kappa, \kappa' \in F$. $\kappa \neq \kappa' \Rightarrow \kappa \notin anc(\kappa')$

We can extend a similar definition to runs $R^{||}$ in the intermediate semantics:

Definition 3 (Intermediate frontiers). Let $R^{||}$ be a run in the intermediate semantics. Then a set F of identifiers is a frontier of $R^{||}$ on blockchain \mathbb{B} (written frontier $(F, R^{||}, \mathbb{B})$) if the following conditions hold:

- 1) $F \subseteq R^{||}$
- 2) $\forall \kappa' \in R^{||}$. $\exists \kappa \in F$. $\kappa \in anc(\kappa') \lor \kappa \in desc(\kappa')$
- 3) $\forall \kappa \kappa' \in F. \ \kappa \neq \kappa' \Rightarrow \kappa \notin anc(\kappa')$

We define an ordering among frontiers.

Definition 4 (Frontier ordering). We say that a frontier F_1 is at least as small as a frontier F_2 (written $F_1 \leq F_2$) if

$$\forall \kappa_2 \in F_2. \ \exists \kappa_1 \in F_1. \ \kappa_1 \in anc(\kappa_2)$$

Similarly, we say that frontier F_1 is smaller than frontier F_2 (written $F_1 < F_2$) if $F_1 \le F_2$ but $F_1 \ne F_2$.

Note that the minimal frontier of a run contains exactly the roots of all contracts appearing in a run.

Definition 5 (Root Contracts). We say that a set F of identifiers are the root contracts of a run R^x (written $roots(R^x, F)$) if the following conditions hold:

- 1) $frontier(F, R^x)$
- 2) $\forall F'$. frontier $(F', R^x) \Rightarrow F \leq F'$

Lemma 3 (Existence of $BitML^x$ root contracts). Let R^x be a (valid) $BitML^x$ run. Then there exists a unique set F of root contracts:

$$\exists F : roots(R^x, F) \land \forall F' : roots(F', R^x) \Rightarrow F = F'$$

We write $F = roots(R^x)$ in this case.

Note that each frontier can be partitioned by their root contracts.

Definition 6 (Maximal Frontier, $BitML^x$). A frontier F of a run R^x is maximal (written $maxFront(R^x, F)$) if the following conditions hold

1) frontier (F, R^x)

2) $\forall F'$. frontier $(F', R^x) \Rightarrow F' < F$

Lemma 4 (Existence of maximal $BitML^x$ frontier). Let R^x be a (valid) $BitML^x$ run. Then there exists a unique maximal frontier:

$$\exists F : maxFront(F, R^x) \land \forall F' : maxFront(F', R^x) \Rightarrow F = F'$$

We write $F = \max Frontier(R^x)$ in this case.

Definition 7 (Maximal Frontier, intermediate). A frontier F of a run $R^{||}$ is maximal (written $maxFront(R^x, F, \mathbb{B})$) if the following conditions hold

- 1) frontier $(F, R^{||}, \mathbb{B})$
- 2) $\forall F'$. frontier $(F', R^{||}, \mathbb{B}) \Rightarrow F' \leq F$

Lemma 5 (Existence of maximal intermediate frontier). Let $R^{||}$ be a (valid) intermediate run and $\mathbb{B} \in \mathcal{B}$. Then there exists a unique maximal frontier for \mathbb{B} :

$$\exists F: \textit{maxFront}(F, R^{||}, \textcolor{red}{\mathbb{B}}) \land \forall F': \textit{maxFront}(F', R^{||}, \textcolor{red}{\mathbb{B}}) \Rightarrow F = F'$$

We write $F = \max Frontier(R^{||})^{\mathbb{B}}$ in this case.

B. Lemmas on Frontiers

Lemma 6 (Frontier configurations). Let R^x be a BitML^x run and F be the maximal frontier of R^x ($F = maxFrontier(R^x)$). Then if $\kappa \in F$ there exist C, B such that $\langle C, B \rangle_{\kappa} \in \Gamma^x(R^x)$.

Helper lemma H1:

$$\kappa \in \Gamma^x(R_0^x) \land \kappa \in \Gamma^x(R^x) \land \mathsf{maxFront}(F_0, R_0^x) \land \mathsf{maxFront}(F, R^x) \implies (\kappa \in F_0 \iff \kappa \in F)$$

Proof: Lemma H1. Induction on R^x with proof by contradiction in both directions.

Proof: Frontier configurations. Proof by Induction on \mathbb{R}^x :

- Base case: $F = \emptyset$
- Inductive case: For every $BitML^x$ transition $R_0^x \xrightarrow{\alpha} R^x$, we define sets $New = \{\kappa \in \Gamma^x(R^x) | \kappa \notin \Gamma^x(R_0^x) \}$, $Deleted = \{\kappa \notin \Gamma^x(R^x) | \kappa \in \Gamma^x(R_0^x) \}$ and $Persistent = \{\kappa \in \Gamma^x(R^x) | \kappa \in \Gamma^x(R_0^x) \}$.

Let F_0 and F be the unique maximal frontiers of R_0^x and R^x (i.e. $\mathsf{maxFront}(F_0, R_0^x)$ and $\mathsf{maxFront}(F, R^x)$). We will show that $F = (F_0 \setminus Deleted) \cup New$ by

- $Deleted \cap F = \emptyset$. Every $BitML^x$ semantics rule that deletes a contract from a configuration, appends a set of descendants to the new configuration:

$$\forall \kappa_0 \in Deleted. \ \exists \kappa \in (desc(\kappa_0) \cap \Gamma^x(R^x))$$

By condition (3) of frontier, we know that for each such pair of $\kappa \in desc(\kappa_0)$ at most one of them can be part of F. By definition of maximality, we conclude that $\kappa_0 \notin F$.

- $New \subseteq F$. By definition of frontiers we know that $\kappa \in New \implies \exists \kappa' \in F$. $\kappa' \in anc(\kappa) \lor \kappa' \in desc(\kappa)$. However κ' cannot be a strict ancestor since F is maximal. Additionally, by $BitML^x$ semantics we know that no descendant of κ is part of run R^x since it was freshly added. Hence, $\kappa \in F$.
- Persistent ⊆ $F_0 \cap F$: By lemma (H1)
- $\not\exists \kappa. \ \kappa \not\in \Gamma^x(R^x) \land \kappa \not\in \Gamma^x(R_0^x) \land \kappa \in F \land \kappa \not\in F_0$

Towards contradiction, we derive that $\kappa \in R_0^x$ i.e. there exists $\kappa' \in desc(\kappa)$ that is in the maximal frontier $(\kappa' \in F_0)$ and the configuration $\Gamma^x(R_0^x)$. Hence, either κ' or its direct descendant is in $\Gamma^x(R^x)$ by $BitML^x$ semantics. This contradict maximality.

To be shown $\forall \kappa \in F$. $\exists C, B$. $\langle C, B \rangle_{\kappa} \in \Gamma^{x}(R^{x})$. Case distinction on κ :

- $\kappa \in Persistent$: By IH
- **–** κ ∈ *New*: By definition of *New*

Lemma 7. Let R^x be a BitML^x run and F be the maximal frontier of R^x ($F = \max Frontier(R^x)$). Then for every $\langle C, \mathbf{B} \rangle_{\kappa} \in \Gamma_{R^x}$, $\kappa \in F$.

On frontiers and coherence. Intuitively, the maximal frontier of a $BitML^x$ run coincides with the frontier derived from the union over the maximal frontiers (per blockchain, by taking their maximum) of a coherent intermediate run on all non-compensated contracts.

Similarly, the maximal frontier of the common prefix coincides with the frontier derived from the union over the maximal frontiers (per blockchain, by taking their minimum) of a coherent intermediate run on all non-compensated contracts. We spell this out more formally in the following.

Definition 8 (Join of intermediate frontiers). Let F_1 and F_2 be frontiers of a run $R^{||}$. Then the join of F_1 and F_2 (written $F_1 \sqcup F_2$) is a set satisfying the following conditions

- 1) $F_1 \sqcup F_2 \subseteq F_1 \cup F_2$
- 2) $\forall \kappa' \in F_1 \cup F_2$. $\exists \kappa \in F_1 \sqcup F_2$. $\kappa \in desc(\kappa')$
- 3) $\forall \kappa \kappa' \in F_1 \sqcup F_2$. $\kappa \neq \kappa' \Rightarrow \kappa \notin anc(\kappa')$

Intuitively, the join of two frontiers is the upper bound of the frontiers.

Note that the join $F_1 \sqcup F_2$ of two frontiers always contains the far most progressed contracts out of the chains. This accounts for both moves that ran ahead, as well as compensated chains.

Lemma 8 (Single-blockchain frontier update). Let $R^{||} = R_0^{||} \xrightarrow{\alpha(\kappa, \mathbb{B})}$ be an intermediate semantics run with the move α resulting in successor contracts $\vec{\kappa}$. Then it holds that:

$$maxFrontier(R^{||}, \mathbb{B}) = maxFrontier(R_0^{||}, \mathbb{B}) \setminus \{\kappa\} \cup \{\kappa^{\downarrow} \in \vec{\kappa}\}$$

Definition 9. We say that an intermediate semantics run $R^{||}$ is "non-divergent" (we denote this nonDivergent $(R^{||})$) if for every pair of moves $\alpha(\kappa, \mathbb{B}), \alpha'(\kappa, \mathbb{B}') \in R^{||}$, with $\alpha, \alpha' \in \{dwithdraw, cwithdraw, reveal, split, right\}$, it holds that $\alpha = \alpha'$.

Lemma 9 (Eventual Synchronicity Frontier Update). Let $R^{||} = R_0^{||} \xrightarrow{\alpha(\kappa, \mathbb{B})}$ be a non-divergent run with the move α resulting in successor contracts $\vec{\kappa}$. Then it holds that:

$$\bigsqcup_{\mathbb{B}_{i} \in \mathcal{B}} \mathit{maxFrontier}(R^{||}, \mathbb{B}_{i}) = \begin{cases} \bigsqcup_{\mathbb{B}_{i} \in \mathcal{B}} \mathit{maxFrontier}(R^{||}_{0}, \mathbb{B}_{i}) \setminus \{\kappa\} \cup \{\kappa^{\downarrow} \in \vec{\kappa}\} & \mathit{if } \kappa \in \bigsqcup_{\mathbb{B}_{i} \in \mathcal{B}} \mathit{maxFrontier}(R^{||}_{0}, \mathbb{B}_{i}) \\ \bigsqcup_{\mathbb{B}_{i} \in \mathcal{B}} \mathit{maxFrontier}(R^{||}_{0}, \mathbb{B}_{i}) & \mathit{otherwise} \end{cases}$$

Proof. We are doing a $\alpha(\kappa, \mathbb{B})$ move resulting in successors $\vec{\kappa}$. We then know that $\kappa \in \mathsf{maxFrontier}(\mathbb{B}, R_0^{||})$ and the updated maximal frontier for blockchain \mathbb{B} will be $\mathsf{maxFrontier}(\mathbb{B}, R^{||}) = \mathsf{maxFrontier}(\mathbb{B}, R_0^{||}) \setminus \{\kappa\} \cup \{\kappa' \in \vec{\kappa}\}$. But, because the $\alpha(\kappa, \mathbb{B})$ only affects blockchain \mathbb{B} , for other blockchains $\hat{\mathbb{B}} \neq \mathbb{B}$, we will have $\mathsf{maxFrontier}(\mathbb{B}', R^{||}) = \mathsf{maxFrontier}(\mathbb{B}', R_0^{||})$. Let's first consider the join of maximal frontiers for all blockchains.

$$\begin{split} & \bigsqcup_{\mathbb{B}' \in \mathcal{B}} \mathsf{maxFrontier}(R^{||}, \mathbb{B}') = \bigsqcup_{\mathbb{B}' \neq \mathbb{B}} \mathsf{maxFrontier}(R^{||}, \mathbb{B}') \sqcup \mathsf{maxFrontier}(\mathbb{B}, R^{||}) \\ & = \bigsqcup_{\mathbb{B}' \neq \mathbb{B}} \mathsf{maxFrontier}(R^{||}_0, \mathbb{B}') \sqcup \left(\mathsf{maxFrontier}(\mathbb{B}, R^{||}_0) \setminus \{\kappa\} \cup \{\kappa' \in \vec{\kappa}\}\right) \end{split}$$

We split by cases on whether κ is up to date in \mathbb{B} .

• If $\operatorname{upToDate}(R_0^{||}, \mathbb{B}, \kappa)$, then $\kappa \in \bigsqcup_{\mathbb{B}' \in \mathcal{B}} \operatorname{maxFrontier}(R_0^{||}, \mathbb{B}')$. We will prove, by applying the definition of the join of frontiers, that:

$$\begin{split} & \bigsqcup_{\mathbb{B}' \neq \mathbb{B}} \mathsf{maxFrontier}(R_0^{||}, \mathbb{B}') \sqcup \left(\mathsf{maxFrontier}(\mathbb{B}, R_0^{||}) \setminus \{\kappa\} \cup \{\kappa' \in \vec{\kappa}\} \right) \\ & = \bigsqcup_{\mathbb{B}' \in \mathcal{B}} \mathsf{maxFrontier}(R_0^{||}, \mathbb{B}') \setminus \{\kappa\} \cup \{\kappa' \in \vec{\kappa}\} \end{split}$$

- 1) For every $\hat{\kappa} \in \bigsqcup_{\mathbb{B}' \in \mathcal{B}} \mathsf{maxFrontier}(R_0^{||}, \mathbb{B}') \setminus \{\kappa\} \cup \{\kappa' \in \vec{\kappa}\}$, we can see that either $\hat{\kappa} \in \bigsqcup_{\mathbb{B}' \in \mathcal{B}} \mathsf{maxFrontier}(R_0^{||}, \mathbb{B}') \setminus \{\kappa\}$ or $\hat{\kappa} \in \vec{\kappa} \subseteq \mathsf{maxFrontier}(\mathbb{B}, R_0^{||}) \setminus \{\kappa\} \cup \{\kappa' \in \vec{\kappa}\}$.
- 2) Let $\kappa' \in \bigsqcup_{\mathbb{B}' \neq \mathbb{B}} \mathsf{maxFrontier}(R_0^{||}, \mathbb{B}') \cup (\mathsf{maxFrontier}(\mathbb{B}, R_0^{||}) \setminus \{\kappa\} \cup \{\kappa' \in \vec{\kappa}\})$. We split by case on whether κ' is an ancestor of κ or not. If $\kappa' \notin anc(\kappa)$, then there is a descendant $\kappa^{\downarrow} \in desc(\kappa')$ in $\bigsqcup_{\mathbb{B}' \neq \mathbb{B}} \mathsf{maxFrontier}(R_0^{||}, \mathbb{B}')$ and it cannot be κ , so $\kappa^{\downarrow} \in \mathsf{maxFrontier}(\mathbb{B}, R_0^{||}) \setminus \{\kappa\} \cup \{\kappa' \in \vec{\kappa}\}$. If instead $\kappa' \in anc(\kappa)$, then all the the successors of κ are also descendants of κ' .
- 3) Let $\kappa_1, \kappa_2 \in \bigsqcup_{\mathbb{B}' \in \mathcal{B}} \mathsf{maxFrontier}(R_0^{||}, \mathbb{B}') \setminus \{\kappa\} \cup \{\kappa' \in \vec{\kappa}\}$. We will prove by contradiction that $\kappa_1 \notin anc(\kappa_2)$. If $\kappa_1 \in anc(\kappa_2)$ then that would imply that either:

- $-\bigsqcup_{\mathbb{B}'\in\mathcal{B}} \mathsf{maxFrontier}(R_0^{||},\mathbb{B}')$ is not a frontier, in the case where both κ_1 and κ_2 are in $\in \bigsqcup_{\mathbb{B}'\in\mathcal{B}} \mathsf{maxFrontier}(R_0^{||},\mathbb{B}')\setminus \{\kappa\}$. We know by inductive hypothesis that $\bigsqcup_{\mathbb{B}'\in\mathcal{B}} \mathsf{maxFrontier}(R_0^{||},\mathbb{B}')$ is a frontier.
- Frontier.

 κ had an ancestor in $\bigsqcup_{\mathbb{B}' \in \mathcal{B}} \mathsf{maxFrontier}(R_0^{||}, \mathbb{B}')$, in the case where $\kappa_2 \in \{\kappa' \in \vec{\kappa}\}$. But we know that $\kappa \in \bigsqcup_{\mathbb{B}' \in \mathcal{B}} \mathsf{maxFrontier}(R_0^{||}, \mathbb{B}')$ and by the definition of frontiers, there cannot be another ancestor.
- If not upToDate $(R_0^{||}, \mathbb{B}, \kappa)$, because we know that $\kappa \in \mathsf{maxFrontier}(R^{||}, \mathbb{B})$ it must be the case that $\kappa \notin \bigsqcup_{\mathbb{B}' \in \mathcal{B}} \mathsf{maxFrontier}(R_0^{||}, \mathbb{B}')$. We will prove, by applying the definition of the join of frontiers, that:

$$\bigsqcup_{\mathbb{B}'\neq\mathbb{B}} \operatorname{maxFrontier}(R_0^{||},\mathbb{B}') \sqcup \left(\underbrace{\max_{\mathbb{B}'\neq\mathbb{B}}}_{\mathbb{B}'} \right) \setminus \{\kappa\} \cup \{\kappa' \in \vec{\kappa}\} \right)$$

$$= \bigsqcup_{\mathbb{B}'\in\mathcal{B}} \operatorname{maxFrontier}(R_0^{||},\mathbb{B}') \sqcup \underbrace{\max_{\mathbb{B}'\neq\mathbb{B}}}_{\mathbb{S}'} \sqcup \underbrace{\min_{\mathbb{B}'\neq\mathbb{B}}}_{\mathbb{S}'} \sqcup \underbrace{\min_{\mathbb{B}'$$

Using the definition of the join of frontiers:

- 1) Let $\hat{\kappa} \in \bigsqcup_{\mathbb{B}' \in \mathcal{B}} \mathsf{maxFrontier}(R_0^{||}, \mathbb{B}') \subseteq \textcircled{1} \cup \textcircled{3}$. Then, either $\hat{\kappa} \in \textcircled{1} \subseteq \textcircled{1} \cup \textcircled{2}$ or 3. If $\hat{\kappa} \in \textcircled{3}$ then, if $\hat{\kappa} \neq \kappa$, we know that $\hat{\kappa} \in \textcircled{2}$. But know, because we are in the case where B was not up to date, that $\kappa \notin \bigsqcup_{\mathbb{B}' \in \mathcal{B}} \mathsf{maxFrontier}(R_0^{||}, \mathbb{B}')$
- 2) Let $\hat{\kappa} \in (0 \cup 2)$. If $\hat{\kappa} \in (0 \cup 2) \subseteq (0 \cup 3)$, then by condition (2) of definition 12, we know that there exists an ancestor $\hat{\kappa}^{\downarrow} \in desc(\hat{\kappa})$ such that $\hat{\kappa}^{\downarrow} \in (0 \cup 3)$.
 - If $\hat{\kappa} \in \mathfrak{D}$, then $\hat{\kappa}$ is a successor of κ . We know that $\kappa \in \mathfrak{J} \subseteq \bigcup_{\mathbb{B}' \in \mathcal{B}} \mathsf{maxFrontier}(R_0^{||}, \mathbb{B}')$. By condition (2) of definition 12, this means that there exist a descendant κ^{\downarrow} of κ in $\bigsqcup_{\mathbb{B}' \in \mathcal{B}} \mathsf{maxFrontier}(R_0^{||}, \mathbb{B}')$. By the definition of frontier, this implies that there is, for some other blockchain $\hat{\mathbb{B}}$ a move $\hat{\alpha}(\kappa, \hat{\mathbb{B}}) \in R^{||}$. But, because we know that $R^{||}$ is non-divergent, we know that $\hat{\alpha} = \alpha$ and resulted in the same successors $\hat{\mathbb{D}}$. Then, κ^{\downarrow} is also a descendant of $\hat{\kappa}$.
- 3) Let $\kappa_1, \kappa_2 \in \bigsqcup_{\mathbb{B}' \in \mathcal{B}} \mathsf{maxFrontier}(R_0^{||}, \mathbb{B}')$, then we already know by condition (3) of definition 12 that $\kappa_1 \notin anc(\kappa_2)$.

C. Stipulation Status

Definition 10 (BitML^x stipulation status). Let R^x be a BitML^x run and $\{G\}C \in R^x$ a contract advertisement in the run. Then, the stipulation status of $\{G\}C$ in R^x is defined as:

$$\textit{stipStatus}^x(R^x, \{G\}C) = \begin{cases} Initialized & \textit{if } desc(R^x, \kappa_0) \neq \emptyset \\ Aborted & \textit{if } desc(R^x, \kappa_0) = \emptyset \land \{G\}C \notin \Gamma^x \\ Advertised & \textit{if } \{G\}C \in \Gamma_{R^x} \end{cases}$$

Definition 11 (Intermediate stipulation status). Let $R^{||}$ be a run in the intermediate semantics $\{G\}C \in R^{||}$ a contract advertisement in the run such that $initSettings(G) = (t_0, \kappa_0)$. Then, the intermediate stipulation status of $\{G\}C$ for blockchain \mathbb{B} in $R^{||}$ is defined as:

$$stipStatus^{\mathbb{B}}(R^{||}, \{G\}C) = \begin{cases} \pi.status & \text{if } \exists \langle C, \pi \rangle_{\kappa_0}^{\mathbb{B}} \in \Gamma_{R^{||}} \\ Initialized & \text{if } ldesc(\kappa_0, \mathbb{B}) \neq \emptyset \\ Refunded & \text{if } rdesc(\kappa_0, \mathbb{B}) \neq \emptyset \\ Advertised & \text{if } \{G\}^{\mathbb{B}}C \in R^{||} \\ & \wedge \forall (A:! \ \mathbb{B} @ \vec{x}) \in deposits(G). \forall \mathbb{B} \in \mathcal{B}. \exists \langle A, v \rangle_{x_i}^{\mathbb{B}} \in \Gamma^{||} \\ DoubleSpent & \text{if } \{G\}^{\mathbb{B}}C \in R^{||} \\ & \wedge \exists (A:! \ \mathbb{B} @ \vec{x}) \in deposits(G). \exists \mathbb{B} \in \mathcal{B}. \ \exists \langle A, v \rangle_{x_i}^{\mathbb{B}} \in \Gamma^{||} \end{cases}$$

Definition 12 (Eventual synchronicity stipulation status). Let $R^{||}$ be a run in the intermediate semantics $\{G\}C \in R^{||}$ a contract advertisement in the run. Then, the eventual synchronicity stipulation status of $\{G\}C$ is defined as:

```
initialized if \exists \mathbb{B} \in \mathcal{B} : stipStatus^{\mathbb{B}}(R^{||}, \{G\}C) = Initialized
\textit{stipStatus}^{es}(R^{||}, \{G\}C) = \begin{cases} & \land \forall \mathbb{B} \in \mathcal{B} : \textit{stipStatus}^{\mathbb{B}}(R^{||}, \{G\}C) \notin \{Refunded, DoubleSpent, Advertised\} \\ & \textit{if} \ (\exists \mathbb{B} \in \mathcal{B} : \textit{stipStatus}^{\mathbb{B}}(R^{||}, \{G\}C) \in \{Refunded, DoubleSpent\} \\ & \lor \forall \mathbb{B} \in \mathcal{B} : \textit{stipStatus}^{\mathbb{B}}(R^{||}, \{G\}C) \in \{Right, Slashed, Compensated\}) \\ & \land \forall \mathbb{B} \in \mathcal{B} : \textit{stipStatus}^{\mathbb{B}}(R^{||}, \{G\}C) \notin \{Initialized\} \end{cases}
                                                                                   Advertised if \forall \mathbb{B} \in \mathcal{B}: stipStatus^{\mathbb{B}}(R^{||}, \{G\}C) \notin \{Initialized, Refunded, DoubleSpent\}
                                                                                                                                                    APPENDIX G
                                                                                                                                ROUND-BASED EXECUTION
```

A. Round-based execution

The execution in the intermediate semantics proceeds in rounds based on the depth of the contract. To make this explicit and to derive guarantees based on the round structure we first define a function to access the time-based round status of a contract κ in an intermediate run $R^{||}$:

Definition 13 (Time-based Round Status). Let $R^{||}$ be an intermediate run ending in $\Gamma^{||}$ and F_R be its set of root contracts $(F_R = roots(R^{||}))$. Let $\kappa \in R^{||}$ and $\kappa_0 \in F_R \cap anc(\kappa)$ be the root contract of κ in $R^{||}$. Let further $advertise(\{G\}C) \in R^{||}$ with $(t_0, \kappa_0) = initSettings(G)$. Then we define the round status of κ in $R^{||}$ (written roundStatus(κ , $R^{||}$)) to be the tuple (r,p) where

$$r := \left\lfloor \frac{t - t_0}{2\hat{\delta}} \right\rfloor$$

$$p := \begin{cases} 0 & \text{if } 2r\hat{\delta} \le t - t_0 < (2r + 1)\hat{\delta} \\ 1 & \text{if } (2r + 1)\hat{\delta} \le t - t_0 < (2r + 2)\hat{\delta} \end{cases}$$

Intuitively, r represents the last execution round (where each round takes $2\hat{\delta}$) and p describes the phase within the current execution round. The phases correspond to the different stages of skipping and compensation.

We make the implications of these phases more explicit with the following property, which we will prove as an invariant of our soundness statement:

Property 1 (Round-based Execution). Let $R^{||}$ be an intermediate run ending in $\Gamma^{||}$ and A an honest user. We say that the round fulfils the Round-Based Execution property (we write this roundBasedExecution $(A, R^{||})$) if the following conditions hold:

- For every contract advertisement $\{G\}^{\mathbb{B}}C \in \Gamma^{||}$ with stipulation time t_0 , the following statements hold:
 - 1) If $t < t_0 + \hat{\delta}$:
 - a) $\mathbf{A} \notin \mathcal{R}_{\kappa_0}^{ss}(R^{||})$ b) $\mathbf{A} \notin \mathcal{R}_{\kappa_0}^{is}(R^{||})$ c) $desc(\kappa_0) = \emptyset$

We say that $\{G\}C$ is in publishing phase.

- 2) If $t \ge t_0 + \hat{\delta}$, then stipStatus $(R_0^{||}, \kappa_0) = DoubleSpent$. We say that $\{G\}C$ is invalidated.
- For every stipulation contract $(C,\pi)_{\kappa_0}^{\mathbb{B}} \in \Gamma_{R^{||}}$ such that $advertise(\{G\}C) \in R^{||}$ with $(t_0,\kappa_0) = initSettings(G)$ the following statements hold:
 - 1) If $t < t_0$, then
 - a) $\pi.status = Stip-Choice$

 - b) $A \in \mathcal{R}_{\kappa_0}^{is}(R^{||}) \Longrightarrow \mathbb{AU}_{\{G\}C}^{A}(R^{||}) = \mathcal{U}_{\kappa_0}(R^{||})$ c) $desc(\kappa_0) \neq \emptyset \lor A \in \mathcal{R}_{\kappa_0}^{is}(R^{||}) \Longrightarrow \mathcal{R}_{\kappa_0}^{is}(R^{||}) = \mathcal{U}_{\kappa}(R^{||})$

We say that κ_0 is in initialization phase.

- 2) If $t_0 < t < t_0 + \hat{\delta}$, then
 - a) $\pi.status \notin \{Stip\text{-}Slashed(A), Stip\text{-}Compensation(A)\}$
 - b) $\mathbf{A} \notin \mathcal{R}^{ss}_{\kappa_0}(R^{||})$
 - $c) \ \operatorname{desc}(\kappa_0) \neq \emptyset \implies \mathcal{R}^{is}_{\kappa_0}(R^{||}) = \mathcal{U}_{\kappa}(R^{||}) \wedge \mathcal{R}^{ss}_{\kappa_0}(R^{||}) \setminus \{ {\color{red} A} \} \neq \emptyset$

We say that κ_0 is in compensation phase.

```
a) \exists B \in \mathcal{U}_{\kappa}(R^{||}) \setminus \{A\} : \pi.status \in \{Stip-Slashed(B), Stip-Compensation(B)\}
       b) \mathbf{A} \notin \mathcal{R}^{ss}_{\kappa_0}(\mathbb{R}^{||})
        c) desc(\kappa_0) = \emptyset
       We say that \kappa_0 is in refund phase.
   4) If t_0 + 2 \times \hat{\delta} < t, then
       a) \exists B \in \mathcal{U}_{\kappa}(R^{||}) \setminus \{A\} : \pi.status = Stip-Compensation(B)
       We say that \kappa_0 is finalized.
• for every active contract \langle C, \pi \rangle_{\kappa_0}^{\mathbb{B}} \in \Gamma_{R^{||}} with round status roundStatus(\kappa, R^{||}) = (r, p), the following statements hold:
   1) If |\kappa| > r, then
       a) \pi.status \in \{Choice, Left\} \lor \exists B \in \mathcal{U}_{\kappa}(R^{||}) : \pi.status = Assigned(B)
       b) rdesc(R^{||}, \kappa) = \emptyset
       We say that \kappa is ahead of its round.
   2) If |\kappa| = r \wedge p = 0, then
        a) \pi.status \notin \{Slashed(A), CompensatedFrom(A)\}
       b) (\forall B \in \mathcal{U}_{\kappa}(R^{||}) : \pi.status \neq Assigned(B)) \implies A \notin \mathcal{R}_{\kappa}^{ss}(R^{||})
       c) rdesc(R^{||}, \kappa) = \emptyset
       d) \pi.status = Left \lor desc(\kappa) \neq \emptyset \implies \mathcal{R}^{ss}_{\kappa_0}(R^{||}) \setminus \{A\} \neq \emptyset
       We say that \kappa is in compensation phase.
   3) If |\kappa| = r \wedge p = 1, then
       a) \pi.status \notin \{Choice, Left, Slashed(A), CompensatedFrom(A)\}
       b) (\forall \mathbf{B} \in \mathcal{U}_{\kappa}(R^{||}).\pi.status \neq \mathbf{Assigned}(\mathbf{B})) \implies \mathbf{A} \notin \mathcal{R}^{ss}_{\kappa}(R^{||})
        c) ldesc(R^{||}, \kappa) \neq \emptyset \implies rdesc(R^{||}, \kappa) = \emptyset \land \exists B \in \mathcal{U}_{\kappa}(R^{||}) \setminus \{A\} : \pi.status = CompensatedFrom(B)
       We say that \kappa is in skipping phase.
   4) If |\kappa| < r, then
       a) \exists B \in \mathcal{U}_{\kappa}(R^{||}) : \pi.status = Assigned(B) \text{ or } \exists B \in \mathcal{U}_{\kappa}(R^{||}) \setminus \{A\} : \pi.status = CompensatedFrom(B)
       b) ldesc(R^{||}, \kappa) \neq \emptyset \implies rdesc(R^{||}, \kappa) = \emptyset
       We say that \kappa is finalized.
```

Corollary 1 (Non-divergence). Let $R^{||}$ be an intermediate run and A be an honest user such and roundBasedExecution(A, $R^{||}$). Let further $\kappa \in R^{||}$ be a contract in the run. Then, for every pair of moves $\alpha(\kappa, \mathbb{B}), \alpha'(\kappa, \mathbb{B}')R^{||}$ in different blockchains for the contract, $\alpha = \alpha'$.

Corollary 2 (No honest compensations). Let $R^{||}$ be an intermediate run and A be an honest user such and $roundBasedExecution(A, R^{||})$. Then,

- For every stipulation contract $\langle C, \pi \rangle_{\kappa_0}^{\mathbb{B}} \in \Gamma_{R^{||}}$, $\pi.status \notin \{Stip Slashed(A), Stip CompensatedFrom(A)\}$
- For every active contract $(C, \pi)_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}$, $\pi.status \notin \{Slashed(A), CompensatedFrom(A)\}$

APPENDIX H $BitML^x$ SOUNDNESS

We will now prove the soundness of $BitML^x$. We start by defining some auxiliary lemmas, and some properties that we will prove as invariants.

The first lemma is a property of intermediate runs and says that if we have stipulation contracts in all blockchains, then the advertisement is fully authorized by all participants and we no longer have contract advertisements in the configuration.

Lemma 10 (Fully Published). Let $R^{||}$ be an intermediate ending in $\Gamma^{||}$ and $\{G\}C$ a contract advertisement with $initSettings(\{G\}C) = (t_0, \kappa_0)$ such that for all blockchains $\mathbb{B} \in \mathcal{B}$, we have an intermediate stipulation contract $\langle C, \pi \rangle_{\kappa_0}^{\mathbb{B}} \in \Gamma_{R^{||}}$. Then, we know that:

```
 \begin{array}{l} \bullet \ \mathbb{A}\mathbb{U}^{\pmb{A}}_{\{G\}C}(R^{||}) = users(G) \\ \bullet \ \forall \mathbb{B} \in \mathcal{B} : \{G\}^{\mathbb{B}}C \notin \Gamma_{R^{||}} \end{array}
```

Proof. By standard induction over $R^{||}$.

3) If $t_0 + \hat{\delta} \le t < t_0 + 2 \times \hat{\delta}$, then

The second lemma says that if there was an authorization move at some point in the run and we still have the contract for that authorization, then we still have the authorization object in the configuration.

Lemma 11 (BitML^x Persistence of Authorizations). Let R^x be a BitML^x run ending in Γ^x and κ a contract in the run. If $\langle \vec{A}: D +> C, \mathbf{B} \rangle_{\kappa} \in \Gamma^{||}$ and $A \in \mathcal{A}_{\kappa}(R^x)$ for some user $A \in \vec{A}$, then $A[\kappa \triangleright D] \in \Gamma^{||}$.

Proof. By standard induction over R^x .

The Configuration Consistency invariant will keep track of the relation of the objects in the intermediate and $BitML^x$ runs.

Property 2 (Configuration Consistency). Let $R^{||}$ be an intermediate ending in $\Gamma^{||}$ and R^x a BitML^x run ending in Γ^x . We say that the runs fulfil the Configuration Consistency condition (we write this conf Consistency $(R^x, R^{||})$) if

- 1) $\forall \{G\}^{\mathbb{B}}C \in \Gamma^{||}: initSettings(G) = (\kappa_0, t_0) \land stipStatus^{es}(R^{||}, \{G\}C) = Advertised \implies \{G\}C \in \Gamma^x$ 2) $\forall \langle C, \pi \rangle_{\kappa_0}^{\mathbb{B}} \in \Gamma^{||}: stipStatus^{es}(R^{||}, \{G\}C) = Advertised \implies \{G\}C \in \Gamma^x: initSettings(G) = (\kappa_0, t_0) \land balance_{\mathbb{B}}(G) = \pi.balance$
- 3) $\forall \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma^{||} : \kappa \in \textit{maxFrontier}(R^x) \implies \langle C, \mathbf{B} \rangle_{\kappa} \in \Gamma^x \wedge \mathbf{B}[\mathbb{B}] = \pi.balance$
- 4) $\forall \{ \stackrel{\sim}{B} : s \# N \} \in \Gamma^{||} : \{ \stackrel{\sim}{B} : s \# N \} \in \Gamma^{x} \}$
- 5) $\forall (B: s \# N) \in \Gamma^{||} : (B: s \# N) \in \Gamma^{x}$
- 6) $\forall B[\# \triangleright \{G\}C] \in \Gamma^{||} : B[\# \triangleright \{G\}C] \in \Gamma^x$

The Timeout Consistency property relates the timeout for status transition of a contract to it's depth and current status.

Property 3 (Timeout Consistency). Let $R^{||}$ be an intermediate run and F_R be its set of root contracts ($F_R = roots(R^{||})$). Let $(C, \pi)_{\kappa_0}^{\mathbb{B}} \in \Gamma_{R^{||}}$ be a stipulation contract. Let further $advertise(\{G\}C) \in R^{||}$ with $(t_0, \kappa_0) = initSettings(G)$. Then it holds that:

$$\pi.time = t_0 + s\hat{\delta}$$

where s = 0 if $\pi.status =$ **Stip-Choice** or 1 otherwise.

Let $(C, \pi)^{\mathbb{B}}_{\kappa} \in \Gamma_{R^{||}}$ be a non-terminal active contract and $\kappa_0 \in F_R \cap anc(\kappa)$ be the root contract of κ in $R^{||}$. Let further $advertise(\{G\}C) \in R^{||}$ with $(t_0, \kappa^*) = initSettings(G)$. Then it holds that:

$$\pi.time = t_0 + (2|\kappa| + s)\hat{\delta}$$

where s = 0 if $\pi.status \in \{Choice, Left\}$ or 1 otherwise.

The Honest Skip invariant says that if a contract is in compensation phase, then the honest user wants to do a $BitML^x$ skipmove.

Property 4 (Honest Skip). Let $R^{||}$ be an intermediate ending in $\Gamma^{||}$ and R^x a BitML^x run. Further let A be an honest user with strategy Σ_A^x . Then, we say that the runs and the user fulfil the Honest Skip property (we write this honest Skip $(A, R^x, R^{||})$) if for every $\langle D + \rangle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma^{||}$ with $(r, p) = roundStatus(\kappa, R^{||})$ such that $|\kappa| = r$ (that is, κ is in compensation phase) it holds that $skip(\kappa) \in \Sigma_A^x(\mathbb{R}^x)$.

The Authorized Left invariant says that whenever we have an active contract in the intermediate state for a started left move and the contract requires the authorization of the honest user, then we have their authorization in the $BitML^x$ run.

Property 5 (Authorized Left). Let $R^{||}$ be an intermediate ending in $\Gamma^{||}$, R^x a BitML^x run ending in Γ^x and A and honest user. We say that the user and the runs fulfil the Authorized Left property (we write this authorized Left $(A, R^x, R^{||})$) if for every $\langle \vec{B}: D +> C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma^{||}$ with $\pi.status = \textbf{Left}$ such that $A \in \vec{B}$ and $\langle \vec{B}: D +> C, B \rangle_{\kappa} \in \Gamma_{R^x}$ it holds that $A[\kappa \triangleright D] \in \Gamma_{R^x}$.

And finally, we prove our $BitML^x$ Soundness theorem.

Lemma 12 (BitML^x Soundness). Let A be an honest user with an eager BitML^x strategy Σ_A^x and an intermediate semantics strategy $\Sigma_{\mathbf{A}}^{||} = \mathcal{S}(\Sigma_{\mathbf{A}}^{x}).$

$$\begin{split} \forall R^{||} \ s.t. \ \Sigma_{\pmb{A}}^{||} &\models R^{||}, \\ \exists R^x : \Sigma_{\pmb{A}}^x &\models R^x \land R^x \sim_{\pmb{A}} R^{||} \\ &\land confConsistency(R^x, R^{||}) \\ &\land honestSkip(\pmb{A}, R^x, R^{||}) \\ &\land authorizedLeft(\pmb{A}, R^x, R^{||}) \\ &\land timeoutConsistency(R^{||}) \\ &\land roundBasedExecution(\pmb{A}, R^{||}) \end{split}$$

Proof. We proceed by induction on $R^{||}$.

Initial Configuration.
$$R^{||} = \Gamma_0^{||} = \langle A_0, v_0 \rangle_{x^0}^{\mathbb{B}_0} | \dots | \langle A_n, v_n \rangle_{x_n}^{\mathbb{B}_n}.$$

With no contracts or advertisements, timeout consistency and roun-based execution are trivial.

We will choose the initial run to be the empty configuration $R^x = \Gamma_0^x = \emptyset$. This run is trivially conforming to the honest user strategy, trivially coherent to $R^{||}$ and trivially fulfilling the Configuration Consistency, Honest Skip and Authorized Left invariants.

The initial configuration is our only base case. We move now to proving the inductive cases of run transitions. For the Honest Skip invariant, we will prove only the case of the time delay δ as all other cases are trivial. Similar for the Authorized Left invariant, where we only prove the ileft case.

$$\underline{\textbf{Advertise}}. \ \ R^{||} = R_0^{||} \xrightarrow{advertise(\{G\}C)} \Gamma^{||} \ \text{and we know by inductive hypothesis that} \ \Sigma_{\textbf{A}}^x \vdash R_0^x \wedge R_0^x \sim_{\textbf{A}} R_0^{||}.$$

We are advertising, simultaneously for every blockchain \mathbb{B} , a new contract $\{G\}^{\mathbb{B}}C$ with a fresh unique id κ_0 and stipulation time t_0 . We know that the contract is well-formed and $A \in G$.

If for the current time t of the intermediate run $t < t_0$, then we will mirror this movement in the $BitML^x$ run and choose $R^x = R_0^x \xrightarrow{advertise(\{G\}C)} \Gamma^x$. Otherwise, we ignore the advertisement and keep $R^x = R_0^x$.

Run Conformance

We know by the intermediate semantics that $\{G\}C$ is well formed, it's secrets are fresh and $A \in G$, so this is a valid $BitML^x$ advertisement.

Coherence

The new advertisment $\{G\}C$ has no authorizations both in the $BitML^x$ and intermediate semantics level and $stipStatus^x(R^x, \{G\}C) = stipStatus^{es}(R^{||}, \{G\}C) = Advertised$.

Configuration Consistency

By mirroring the intermediate move $advertise(\{G\}C)$ in the $BitML^x$ run, we add the advertisement in both levels, fulfilling the invariant condition.

Round-Based Execution

We know by the rule for advertisements that $\{G\}C$ is well-formed. This means in particular that for it's advertisement time t_0 and the current time t of $R_0^{||}$, it holds that $t < t_0$

Stipulation Double Spend.
$$R^{||} = R_0^{||} \xrightarrow{double Spend(\{G\}^{\mathbb{B}}C, B)} \Gamma^{||}$$
 and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \land R_0^x \sim_A R_0^{||}$.

User B chose to double-spend his deposit $\langle B, v \rangle_x^{\mathbb{B}} \in \Gamma_{R_0^{||}}$ which is required by the preconditions G. If he is the first to do so, he is invalidating the contract advertisement, which is now aborted.

This move will change the intermediate stipulation status of $\{G\}C$ from stipStatus $(R_0^{||}, \{G\}C) = Advertised$ to stipStatus $(R_0^{||}, \{G\}C) = DoubleSpent$.

We split by cases on the stipulation status for proving Run Conformance and Coherence.

- We first prove by contradiction that $\underline{\mathsf{stipStatus}}^{es}(R_0^{||}, \{G\}C) \neq Initialized$. Suppose that the opposite is true and $\underline{\mathsf{stipStatus}}^{es}(R_0^{||}, \{G\}C) = Initialized$. We split by cases on the round status of $\{G\}C$.
 - If $t < t_0 + \hat{\delta}$, then $desc(\kappa_0) = \emptyset$ for the unique id κ_0 of G. But, if $stipStatus^{es}(R_0^{||}, \{G\}C) = Initialized$ then $\exists \hat{\mathbb{B}} \in \mathcal{B} : stipStatus^{\hat{\mathbb{B}}}(R^{||}, \{G\}C) = Initialized$, which in turn means that $ldesc(\kappa_0, \mathbb{B}) \subseteq desc(\kappa_0) \neq \emptyset$
 - If $t \geq t_0 + \hat{\delta}$, then stipStatus $(R_0^{||}, \{G\}C) = DoubleSpent$ (that is, some other user double-spent already). But if stipStatus $(R_0^{||}, \{G\}C) = Initialized$ then $\forall \mathbb{B} \in \mathcal{B}$: stipStatus $(R_0^{||}, \{G\}C) \neq DoubleSpent$.

• If $\underline{\text{stipStatus}}^{es}(R_0^{||}, \{G\}C) = Advertised$, then this is the first double-spent deposit. We will choose the $BitML^x$ run $R^x = R_0^x \xrightarrow{abort(\kappa_0)} \Gamma^{||}$.

Run Conformance

Because we are doing a $doubleSpend(\{G\}^{\mathbb{B}}C, B)$ move, we know that we have an intermediate contract advertisement $\{G\}^{\mathbb{B}}C$ in $\Gamma_{R_0^{||}}$. We know by the Configuration Consistency invariant (because stipStatus $^{es}(R_0^{||}, \{G\}C) = Advertised$), that $\{G\}C \in \Gamma_{R_0^x}$.

Given that this is the only precondition for a $BitML^x$ $abort(\{G\}C)$ move, R^x is conformant to A's strategy.

Coherence

Given that $\langle B, v \rangle_x^{\mathbb{B}} \notin \Gamma^{||}$, by definition we have stipStatus $(R^{||}, \{G\}C) = DoubleSpent$ and stipStatus $(R^{||}, \{G\}C) = Aborted$. Meanwhile, in R^x the abort move will consume the advertisement $\{G\}C$ and generate no descendants, so we also have stipStatus $(R^x, \{G\}C) = Aborted$.

Other conditions are trivially held by inductive hypothesis, so we can claim that $R^x \sim_A R^{||}$.

Configuration Consistency

Holds for $\{G\}C$ because stipStatus^{es} $(R^{||}, \{G\}C) \neq Advertised$ and for the successor right-descendant contracts, the $abort(\{G\}C)$ move will introduce the assigned contracts in the $BitML^x$ configuration too.

• If $\underline{\mathsf{stipStatus}}^{es}(R_0^{||}, \{G\}C) = Aborted$, then we keep $R^x = R_0^x$, which is trivially conformant, coherent and configuration consistent by inductive hypothesis.

Round-based execution is trivial for this case.

<u>AuthCommit.</u> $R^{||} = R_0^{||} \xrightarrow{commit(B, \{G\}C)} \Gamma^{||}$ and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \land R_0^x \sim_A R_0^{||}$.

A user B is committing, simultaneously for every blockchain B, to it's secrets in the contract advertisement $\{G\}^{B}C$. This includes the step secrets, the init secret and the stipulation step secret.

If for the current time t of the intermediate run $t < t_0$, then we will mirror this movement in the $BitML^x$ run and choose $R^x = R_0^x \xrightarrow{commit(B, \{G\}C)} \Gamma^x$. Otherwise, we ignore the move and keep $R^x = R_0^x$. In both cases, Coherence, round-bases execution and timeout consistency are trivial.

Run Conformance

For the case where B = A, we know by the honest user strategy that if $commit(A, \{G\}C) \in \Sigma_A^{|I|}(R_0^{|I|})$ then $commit(A, \{G\}C\Sigma_A^x(R_0^x))$.

Configuration Consistency

By mirroring the intermediate move $commit(B, \{G\}C)$ in the $BitML^x$ run, we add the secret commitments in both levels, fulfilling the invariant condition.

Stipulation Authorization. $R^{||} = R_0^{||} \xrightarrow{authInit(A,\{G\}^{\mathbb{B}}C)} \Gamma^{||}$ and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \land R_0^x \sim_A R_0^{||}$.

We need to first distinguish by cases on the stipulation status of $\{G\}C$. We know that $stipStatus^{es}(R^{||}, \{G\}C) = Initialized$ because that would imply that $stipStatus^{\mathbb{B}}(R^{||}, \{G\}C) \neq Advertised$ but $\{G\}^{\mathbb{B}}C)\Gamma_{R^{||}}$ as a precondition for the advertisement authorization move. If $stipStatus^{es}(R^{||}, \{G\}C) = Aborted$ then we will ignore this transition and keep $R^x = R_0^x$, which is trivially conforming and (given that the stipulation status doesn't change) coherent.

If stipStatus^{es} $(R^{||}, \{G\}C) = Advertised$ then, similar to active contract authorizations, we will use different tactics if the authorization is from the honest user or the potentially dishonest ones.

• If B = A, we need to further distinguish whether this is the first blockchain where the honest user is authorizing the move or not.

- If $\underbrace{A \notin \mathcal{A}_{\{G\}C}(R_0^{||})}_{R^x}$ then we will mirror this authorization in the $BitML^x$ run. That is, we will choose $R^x = R_0^x \xrightarrow{A: \{G\}C} \Gamma^{||}$.

Run Conformance

We first need to prove that $A: \{G\}C$ is a valid move in R_0^x . For this, the $BitML^x$ semantics requires that we have a contract advertisement $\{G\}C \in \Gamma_{R_0^x}$ and that we have all of the needed secret commitments.

Because we are doing an $authInit(A, \{G\}^{\mathbb{B}}C)$ move, we know we have a the secret commitments $C[\# \triangleright \{G\}C] \in \Gamma_{R_0^{||||}}$ for every participant $C \in P$, and with our Configuration Consistency invariant we can also claim that $C[\# \triangleright \{G\}C] \in \Gamma_{R_0^n}$.

Finally, we know this authorization to be conforming to the $BitML^x$ strategy of A by the definition of our strategy compilation that states that we only start an authorization in the intermediate semantics level when the user is doing so in the $BitML^x$ level.

Coherence

We need to prove that after the new authorization, $\mathcal{A}_{\{G\}C}(R^x) = \mathbb{AU}_{\{G\}C}^A(R^{||}) \cup (\mathcal{A}_{\{G\}C}(R^{||}) \cap \{A\})$ still holds. We first observe that $\mathcal{A}_{\{G\}C}(R^{||}) = \mathcal{A}_{\{G\}C}(R_0^{||}) \cup \{A\}$. Then, we can do the following reasoning, which is the same as in the case for active contract authorizations:

$$\begin{split} \mathcal{A}_{\{G\}C}(R^x) &= \mathcal{A}_{\{G\}C}(R_0^x) \cup \{A\} \\ & \mathbb{A}\mathbb{U}_{\{G\}C}^A(R_0^{||}) \cup (\mathcal{A}_{\{G\}C}(R_0^{||}) \cap \{A\}) \cup \{A\} \\ & \mathbb{A}\mathbb{U}_{\{G\}C}^A(R_0^{||}) \cup (\mathcal{A}_{\{G\}C}(R_0^{||}) \cup \{A\}) \cap \{(A\} \cup A) \\ & \mathbb{A}\mathbb{U}_{\{G\}C}^A(R_0^{||}) \cup (\mathcal{A}_{\{G\}C}(R^{||}) \cap \{A\}) \end{split}$$

- If $\underline{A} \in \mathcal{A}_{\{G\}C}(R_0^{||})$, then we don't extend the run and keep $R^x = R_0^x$, which is trivially conforming. To see that it's also coherent, we can do:

$$\begin{split} \mathcal{A}_{\{G\}C}(R^x) &= \mathcal{A}_{\{G\}C}(R_0^x) \\ &= \mathbb{A}\mathbb{U}_{\{G\}C}^{A}(R_0^{||}) \cup (\mathcal{A}_{\{G\}C}(R_0^{||}) \cap \{A\}) \\ &= \mathbb{A}\mathbb{U}_{\{G\}C}^{A}(R_0^{||}) \cup ((\mathcal{A}_{\{G\}C}(R_0^{||}) \cup \{A\}) \cap \{A\}) \\ &= \mathbb{A}\mathbb{U}_{\{G\}C}^{A}(R_0^{||}) \cup (\mathcal{A}_{\{G\}C}(R^{||}) \cap \{A\}) \end{split}$$

- If $\underline{B \neq A}$, we need to further distinguish whether this is the last blockchain where the potentially malicious user is authorizing the contract or not, up to compensations for \underline{A} .
 - If $\underline{\underline{B}} \in \mathbb{AU}^{\underline{A}}_{\{G\}C}(R^{||})$, then we extend the $BitML^x$ run with the corresponding authorization. We will choose $R^x = R^x_0 \xrightarrow{\underline{B}: \{G\}C} \Gamma^{||}$.

Conformance

Idem for the case of A, except that this time we don't care about conforming to the strategy, because it's not an honest user.

Coherence

We need to prove that $\mathcal{A}_{\{G\}C}(R^x) = \mathbb{A}\mathbb{U}^{\mathbf{A}}_{\{G\}C}(R^{||}) \cup (\mathcal{A}_{\{G\}C}(R^{||}) \cap \{A\})$. We start by observing that \mathbf{B} is finishing the authorization in this action, so $\mathbb{A}\mathbb{U}^{\mathbf{A}}_{\{G\}C}(R^{||}) = \{B\} \cup \mathbb{A}\mathbb{U}^{\mathbf{A}}_{\{G\}C}(R^{||})$. Then, we have:

$$\begin{split} \mathcal{A}_{\{G\}C}(R^x) &= \{ {\color{red} B} \} \cup \mathcal{A}_{\{G\}C}(R_0^x) \\ &= \{ {\color{red} B} \} \cup \mathbb{A}\mathbb{U}_{\{G\}C}^A(R_0^{||}) \cup (\mathcal{A}_{\{G\}C}(R_0^{||}) \cap \{ {\color{red} A} \}) \\ &= \mathbb{A}\mathbb{U}_{\{G\}C}^A(R^{||}) \cup (\mathcal{A}_{\{G\}C}(R^{||}) \cap \{ {\color{red} A} \}) \end{split}$$

- If $B \notin \mathbb{AU}^{A}_{\{G\}C}(R^{||})$, then we don't extend the run and keep $R^x = R^x_0$, which is trivially conforming. To see that it's also coherent, we can do:

$$\begin{split} \mathcal{A}_{\{G\}C}(R^x) &= \mathcal{A}_{\{G\}C}(R_0^x) \\ &= \mathbb{A}\mathbb{U}_{\{G\}C}^A(R_0^{||}) \cup (\mathcal{A}_{\{G\}C}(R_0^{||}) \cap \{A\}) \\ &= \mathbb{A}\mathbb{U}_{\{G\}C}^A(R^{||}) \cup (\mathcal{A}_{\{G\}C}(R^{||}) \cap \{A\}) \end{split}$$

Other invariants are trivial in this case.

<u>Publish.</u> $R^{||} = R_0^{||} \xrightarrow{publish(\{G\}^{\center{s}}C)} \Gamma^{||}$ and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \land R_0^x \sim_A R_0^{||}$.

We are doing a $publish(\{G\}^{\mathbb{B}}C)$ move that will consume a contract advertisement $\{G\}^{\mathbb{B}}C$ and introduce a stipulation contract $(C, \pi)_{\kappa_0}^{\mathbb{B}}$ in status $\pi.status =$ **Stip-Choice**.

We do not extend the $BitML^x$ and instead keep $R^x = R_0^x$, so Run Conformance is trivial. For Coherence and configuration cosnsitency, we split by cases in the eventual synchronicity stipulation status of $\{G\}C$.

- We first prove by contradiction that $\underline{\mathsf{stipStatus}}^{es}(R_0^{||}, \{G\}C) \neq Initialized$. The proof is analogous to the *doubleSpend* case.
- If $stipStatus^{es}(R_0^{||}, \{G\}C)in\{Advertised, Aborted\}$, then the move will change the intermediate stipulation status in from $stipStatus^{\mathbb{B}}(R_0^{||}, \{G\}C) = Advertised$ to $stipStatus^{\mathbb{B}}(R^{||}, \{G\}C) = Stip-Choice$ which will not change the eventual synchronicity stipulation status.

Configuration Consistency

Because we are introducing a stipulation contract $(C, \pi)_{\kappa_0}^{\mathbb{B}}$ in the intermediate configuration and its advertisement has stipulation status $\mathrm{StipStatus}^{\mathbb{B}}(R_0^{||}, \{G\}C) = Advertised$, we need to prove that $\{G\}C \in \Gamma_{R^x}$. But because $\{G\}^{\mathbb{B}}C \in \Gamma_{R_0^{||}}$, we know by inductive hypothesis that $\{G\}C \in \Gamma_{R_0^x} = \Gamma_{R^x}$.

Round-Based Execution

We know by inductive hypothesis that $t < t_0 + \hat{\delta}$. This means that the new stipulation contract κ_0 can either be in initialization phase or in compensation phase. In both cases, the status $\pi.status = \mathbf{Stip\text{-}Choice}$ is consistent. We also know by the inductive hypothesis that $\mathbf{A} \notin \mathcal{R}^{ss}_{\kappa_0}(R^{||})$. If $t \geq t_0$, then we know that $desc(\kappa_0) = \emptyset$ because (again by inductive hypothesis) $\mathbf{A} \notin \mathcal{R}^{is}_{\kappa_0}(R^{||})$.

Timeout Consistency

Follows directly from the intermediate rule for init.

Init secret reveal. $R^{||} = R_0^{||} \xrightarrow{B:is_{\kappa_0}^B} \Gamma^{||}$ and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \land R_0^x \sim_A R_0^{||}$.

Some user B is revealing their init secret for stipulation contract κ_0 .

We do not extend the $BitML^x$ and instead keep $R^x = R_0^x$. Since this intermediate move doesn't change the frontiers or authorizations, Run Conformance and Coherence are trivial. The Honest Skip and Authorized Left invariants are also trivial in this case.

Round-Based Execution

Let $adv(\kappa_0) = \{G\}C$. If B = A, we know by our strategy compilation that $\forall \mathbb{B} \in \mathcal{B} : \exists \langle C, \pi \rangle_{\kappa_0}^{\mathbb{B}}$. Then, by the fully published lemma, we also know that $\mathbb{AU}^{A}_{\{G\}C}(R^{||}) = users(G)$ which fulfils the stipulation contract condition and $\forall \mathbb{B} \in \mathcal{B} : \{G\}^{\mathbb{B}}C \notin \Gamma_{R^{||}}$ which makes the contract advertisement condition trivial.

Stipulation step secret reveal. $R^{||} = R_0^{||} \xrightarrow{B: s_{\kappa_0}^B} \Gamma^{||}$ and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \land R_0^x \sim_A R_0^{||}$.

Some user B is revealing their step secret for stipulation contract κ_0

We do not extend the $BitML^x$ and instead keep $R^x = R_0^x$. Since this intermediate move doesn't change the frontiers or authorizations, Run Conformance and Coherence are trivial. The Honest Skip and Authorized Left invariants are also trivial

in this case.

Round-Based Execution

If B = A, we know by our strategy compilation that $t < t_0$ where t_0 is the stipulation time of contract κ_0 , so the contract is in initialization phase. But we also know by the strategy compilation that $\mathcal{R}^{is}_{\kappa_0}(R^{||}) = users(\kappa_0)$, that is, all users have revealed their init secrets.

Stipulation Init. $R^{||} = R_0^{||} \xrightarrow{init(\{G\}C,\mathbb{B})} \Gamma^{||}$ and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \land R_0^x \sim_A R_0^{||}$.

We are doing an $init(\kappa_0, \mathbb{B})$ move in the intermediate semantics that will consume a stipulation contract $\langle C, \pi \rangle_{\kappa_0}^{\mathbb{B}}$ with $\pi.status =$ **Stip-Choice** and introduce a new active contract $\langle C, \pi' \rangle_{\kappa}^{\mathbb{B}}$ with $\pi'.status =$ **Choice**, $\pi'.time = \pi.time + 2\hat{\delta}$ and $\kappa = [\kappa_0]$.

Intuitively, we are initializing the contract in blockchain B. This means that all participants, (A included), have revealed their stipulation secrets. We can assume then that the contract has been authorized and published in all blockchains.

For proving $BitML^x$ Run Conformance and Coherence, we will split by cases on the eventual synchronicity stipulation status of κ_0 .

- We prove first by contradiction that $\underline{\text{stipStatus}}^{es}(R_0^{||}, \{G\}C) \neq Aborted$. Suppose the opposite is true and $\underline{\text{stipStatus}}^{es}(R_0^{||}, \{G\}C) = Aborted$. We split by cases according to the definition:
 - If $\exists \hat{\mathbb{B}} \in \mathcal{B}$: stipStatus $\hat{\mathbb{B}}(R^{||}, \{G\}C) = Refunded$, then according to the round-based execution invariant, $t \geq t_0 + \hat{\delta}$. But, because $\pi.status =$ Choice, we know that $t < t_0 + \hat{\delta}$
 - If $\exists \hat{\mathbb{B}} \in \mathcal{B}$: stipStatus $(R^{||}, \{G\}C) = DoubleSpent$, then $\{G\}^{\hat{\mathbb{B}}}C \in \Gamma_{R_0^{||}}$ and round-based execution says that $A \notin \mathcal{R}_{\kappa_0}^{is}(R_0^{||})$. But, according to the intermediate semantics rule for init, we require that $A \in \mathcal{R}_{\{G\}C}^{is}(R_0^{||})$.
 - If $\forall \hat{\mathbb{B}} \in \mathcal{B}$: stipStatus $(R^{||}, \{G\}C) \in \{Right, Slashed, Compensated\}$, then this is also true in particular for our currently initializing blockchain \mathbb{B} . But, according to the intermediate semantics rule for *init*, we require that $\pi.status \neq \mathbf{Stip-Choice}$.
- If $\frac{\text{stipStatus}^{es}(R_0^{||}, \{G\}C) = Advertised}{R^x = R_0^x}$, then this is the first blockchain to initialize. We will choose

Run Conformance

Because we are doing an $init(\kappa_0,\mathbb{B})$ move, we know by the intermediate semantics that we have a stipulation contract $\langle C,\pi\rangle_{\kappa_0}^\mathbb{B}\in\Gamma_{R_0^{|||}}$. Then, by the Configuration Consistency invariant, we know that $\{G\}C\in\Gamma_{R_0^x}$. Similarly, we know that for all users $B\in users(G):B[\#\triangleright\{G\}C]\in\Gamma_{R_0^{|||}}$ so we also have $B[\#\triangleright\{G\}C]\in\Gamma_{R_0^x}$. Finally, we need to prove that we have all the needed authorizations. Again, because we are doing a $init(\kappa_0,\mathbb{B})$, we know that A has revealed $is_{\kappa_0}^A$. According round-based execution, this means that in the intermediate semantics, all users have fully authorized, formally $\mathbb{AU}_{\{G\}C}^A(R^{||})=\mathcal{U}_{\kappa_0}(R^{||})$. Then, from the Coherence invariant, we get that

$$\mathcal{A}_{\{G\}C}(R^x) = \mathbb{AU}_{\{G\}C}^{A}(R^{||}) \cup (\mathcal{A}_{\{G\}C}(R^{||}) \cap \{A\}) = \mathcal{U}_{\{G\}C}(R^{||})$$

In conclusion, $init(\{G\}C)$ is a valid move in the $BitML^x$ semantics and so $\Sigma_A^x \vdash R^x$.

Coherence

This $init(\kappa_0,\mathbb{B})$ move will change the intermediate stipulation status of $\{G\}C = adv(R^{||},\kappa_0)$ from stipStatus $(R_0^{||},\{G\}C) = \mathbf{Stip-Choice}$ to stipStatus $(R_0^{||},\{G\}C) = \mathbf{Initialized}$. We also know by inductive hypothesis, because stipStatus $(R_0^{||},\{G\}C) = \mathbf{Advertised}$, that for all other blockchains $(R_0^{||},\{G\}C) \notin \{Refunded, DoubleSpent\}$. By definition we then have that now stipStatus $(R_0^{||},\{G\}C) = \mathbf{Initialized}$. Similarly, because we are introducing an active contract $[\kappa_0] \in desc(R^x,\kappa_0)$, we also have stipStatus $(R^x,\kappa_0) = \mathbf{Initialized}$. Other conditions are trivially held by inductive hypothesis, so we can claim that $(R^x \sim_A R^{||})$.

Configuration Consistency

Because we are adding a new active contract $\langle C, \pi \rangle_{[\kappa_0]}^{\mathbb{B}}$ to the intermediate configuration and $[\kappa_0] \in \mathsf{maxFrontier}(R^x)$, we need to prove that $\langle C, \mathsf{B} \rangle_{\kappa} \in \Gamma^x$ and $\mathsf{B}[\mathbb{B}] = \pi.balance$.

But, because $(C,\pi)_{\kappa_0}^{\mathbb{B}} \in \Gamma_{R_0^{||}}$ and $\operatorname{stipStatus}^{es}(R_0^{||},\{G\}C) = Advertised$ we know by inductive hypothesis that $\{G\}C \in \Gamma^x$ and $balance_{\mathbb{B}}(G) = \pi.balance$. And according to the $BitML^x$ semantics, the $init(\{G\}C)$ move, will introduce into the configuration $(C,\mathbb{B})_{\kappa}$ with $balance(G) = \mathbb{B}$.

• If stipStatus^{es} $(R_0^{||}, \{G\}C) = Initialized$, then this is not the first blockchain to initialize. We will not extend the $Bit\overline{ML^x}$ run and instead keep $R^x = R_0^x$, which is trivially conformant to the honest user strategy.

Coherence

We know by inductive hypothesis that $\operatorname{stipStatus}^x(R_0^x,\kappa_0) = \operatorname{stipStatus}^{es}(R_0^{||},\{G\}C) = \operatorname{Initialized}$. On the intermediate semantics side, we are doing an $\operatorname{init}(\kappa_0,\mathbb{B})$ move, which will not change the status so $\operatorname{stipStatus}^{es}(R^{||},\{G\}C) = \operatorname{stipStatus}^{es}(R_0^{||},\{G\}C)$. And we also defined $R^x = R_0^x$, so we can conclude that $\operatorname{stipStatus}^x(R^x,\{G\}C) = \operatorname{stipStatus}^{es}(R^{||},\{G\}C)$.

Again, other conditions are trivially held by inductive hypothesis, so we can claim that $R^x \sim_A R^{||}$.

Timeout Consistency

Because $\pi.status =$ Stip-Choice, we know by inductive hypothesis that $\pi.time = t_0$ where $advertise(\{G\}C) \in R^{||}$ and $(t_0, \kappa^*) = initSettings(G)$. Then, for the new status π' after the move, we can reason in the following way:

$$\pi'.time = \pi.time + 2\hat{\delta} = t_0 + 2|\kappa|\hat{\delta}$$

Round-Based Execution

Because $\pi.status =$ Choice, we know that $t < t_0 + \hat{\delta}$ or, equivalently $t - t_0 < \hat{\delta}$. If the round status of the new active contract κ is $(r,p) = \text{roundStatus}(\kappa,R^{||})$ then $r = \left\lfloor \frac{t-t_0}{2\hat{\delta}} \right\rfloor = 0 < |\kappa|$. That is, κ is waiting for synchronisation, which is consistent with the status $\pi.status =$ Choice.

Stipulation Right.

$$R^{||} = R_0^{||} \xrightarrow{sright(\{G\}C,\mathbb{B})} \Gamma^{||}$$
 and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \land R_0^x \sim_A R_0^{||}$.

We are doing an intermediate $sright(\kappa, \mathbb{B})$ move that will change the internal state of a stipulation contract $\langle C, \pi \rangle_{\kappa_0}^{\mathbb{B}}$ in status $\pi.status =$ **Stip-Choice** to a new state with $\pi'.status =$ **Stip-Right** and $\pi'.time = \pi.time + \hat{\delta}$.

Intuitively, we are skipping initialization on blockchain \mathbb{B} . This can be either an asynchronous initialization by the adversary, or part of a regular abort move.

For proving Run Conformance, Coherence and Configuration Consistency, we will split by cases on the status of the stipulation of κ_0 in $R^{||}$.

- If $\underline{\text{stipStatus}}^{es}(R_0^{||}, \{G\}C) = Advertised$, then we need to further distinguish the case where this is the last blockchain to move right, which corresponds to aborting the contract, or not.
 - If $\forall \hat{\mathbb{B}} \in \mathcal{B}$: stipStatus $(R^{||}, \{G\}C) \in \{Right, Slashed, Compensated\}$, then \mathbb{B} was the last to move right. We will choose the $BitML^x$ run $R^x = R_0^x \xrightarrow{abort(\{G\}C)}$.

Run Conformance

Because we are doing a $sright(\kappa_0,\mathbb{B})$ move, we know that we have a stipulation contract $(C,\pi)_{\kappa_0}^{\mathbb{B}}$ in $\Gamma_{R_0^{||}}$. And because stipStatus $^{es}(R_0^{||},\{G\}C)=Advertised$, we know by the Configuration Consistency invariant, that $\{G\}C\in\Gamma_{R_0^x}$. Given that this is the only precondition for an $abort(\{G\}C)$, R^x is conformant to A's strategy.

Coherence

Given that $\forall \hat{\mathbb{B}} \in \mathcal{B}: \text{stipStatus}^{\hat{\mathbb{B}}}(R^{||}, \{G\}C) \in \{Right, Slashed, Compensated\}$, by definition we have stipStatus $^{es}(R^{||}, \{G\}C) = Aborted$. Meanwhile, because we defined our $BitML^x$ run as $R^x = R_0^x \xrightarrow{abort(\{G\}C)}$, the

move will consume the contract advertisement and produce no descendants, so we also have $stipStatus^x(R^x, \{G\}C) = Aborted$.

Other conditions are trivially held by inductive hypothesis, so we can claim that $R^x \sim_A R^{||}$.

- If $\exists \hat{\mathbb{B}} \in \mathcal{B}$: stipStatus $(R^{||}, \{G\}C) \notin \{Right, Slashed, Compensated\}$, then we keep $R^x = R_0^x$, which is trivially conformant and coherent by inductive hypothesis.
- If stipStatus $^{es}(R_0^{||}, \{G\}C) = \{Aborted, Initialized\}$, then we also keep $R^x = R_0^x$, which is again trivially conformant and coherent by inductive hypothesis.

Timeout Consistency

Because $\pi.status =$ **Stip-Choice**, we know by inductive hypothesis that $\pi.time = t_0$ where $advertise(\{G\}C) \in R^{||}$ and $(t_0, \kappa^*) = initSettings(G)$. Then, for the new status π' after the move, we can reason in the following way:

$$\pi'.time = \pi.time\hat{\delta} = t_0 + \hat{\delta}$$

Round-Based Execution

Because $\pi.status =$ Choice, we know that $t < t_0 + \hat{\delta}$. But we also know, by the intermediate semantics rule for sright that $t \ge t_0$, so the contract is in compensation phase, which is consistent with the new status $\pi'.status =$ Stip-Right. Other conditions hold by inductive hypothesis.

Stipulation Abort.

$$R^{||} = R_0^{||} \xrightarrow{abort(\{G\}C,\mathbb{B})} \Gamma^{||}$$
 and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \land R_0^x \sim_A R_0^{||}$.

We are doing an intermediate $abort(\kappa, \mathbb{B})$ move that will consume the stipulation contract $(C, \pi)^{\mathbb{B}}_{\kappa_0}$ in status $\pi.status =$ **Right** and introduce terminal contracts $\kappa_0, \ldots, \kappa_n$ with status $\pi'.status =$ **Stip-Refunded** (A_i) for each user $A_i \in users(G)$.

We are aborting the contract in blockchain \mathbb{B} . This means that we are not finalized at this stage, so there's either some blockchains where the contract was not published, or the dishonest user refused to reveal their init secret. We will split by cases on the status of the stipulation of κ in $R^{||}$.

For proving Run Conformance, Coherence and Configuration Consistency, we will split by cases on the status of the stipulation of κ_0 in $R^{||}$.

- We first prove by contradiction that $\underline{stipStatus}^{es}(R_0^{||}, \{G\}C) \neq Initialized$. Suppose that $\underline{stipStatus}^{es}(R_0^{||}, \{G\}C) = Initialized$. By definition this means that there is some other blockchain $\hat{\mathbb{B}}$ for which $\underline{stipStatus}^{\hat{\mathbb{B}}}(R^{||}, \{G\}C) = Initialized$ and consequently $desc(\kappa_0) \neq \emptyset$. But, we also know, because we are doing an $abort(\kappa_0, \mathbb{B})$ move in the intermediate semantics, that $t \geq t_0 + \hat{\delta}$. And according to the round-based execution invariant, this means that κ_0 is in refund phase, so $desc(\kappa_0) = \emptyset$
- If $\underline{\text{stipStatus}}^{es}(R_0^{||}, \{G\}C) = Advertised$, then this is the first blockchain to abort. We will choose $R^x = R_0^x \xrightarrow{abort(\kappa_0, \mathbb{B})}$.

Run Conformance

Because we are doing an $abort(\kappa_0,\mathbb{B})$ move, we know that we have a stipulation contract $(C,\pi)^{\mathbb{B}}_{\kappa_0}$ in $\Gamma_{R_0^{||}}$. We know by the Configuration Consistency invariant (because stipStatus^{es} $(R_0^{||},\{G\}C)=Advertised$), that $\{G\}C\in\Gamma_{R_0^x}$. Given that this is the only precondition for an $abort(\kappa_0)$, R^x is conformant to A's strategy.

Coherence

This move will instroduce right-descendant contracts, so $stipStatus^{\mathbb{B}}(R^{||}, \{G\}C) = Refunded$. And we know by inductive hypothesis that $\forall \mathbb{B} \in \mathcal{B} : stipStatus^{\mathbb{B}}(R^{||}, \{G\}C) \neq Initialized$, so by definition $stipStatus^{es}(R^{||}, \{G\}C) = Aborted$.

Meanwhile, the $BitML^x$ move will consume the contract advertisement $\{G\}C$, so stipStatus $^x(R^x, \kappa_0) = Aborted$. Other conditions are trivially held by inductive hypothesis, so we can claim that $R^x \sim_A R^{||}$.

Configuration Consistency

Holds for $\{G\}C$ because stipStatus^{es} $(R^{||}, \{G\}C) \neq Advertised$ and for the successor right-descendant contracts, the $abort(\{G\}C)$ move will introduce the assigned contracts in the $BitML^x$ configuration too.

• If $\underline{\text{stipStatus}}^{es}(R_0^{||}, \{G\}C) = Aborted$, then we have already aborted the contract. We will not extend the $BitML^x$ run and instead keep $R^x = R_0^x$, which is trivially conformant to the honest user strategy.

Coherence

We know by inductive hypothesis that $\operatorname{stipStatus}^x(R_0^x,\kappa_0)=\operatorname{stipStatus}^{es}(R_0^{||},\{G\}C)=Aborted$. On the intermediate semantics side, we are moving the intermediate stipulation status in $\mathbb B$ from $\operatorname{stipStatus}^{\hat{\mathfrak B}}(R_0^{||},\{G\}C)=\operatorname{Stip-Right}$ to $\operatorname{stipStatus}^{\hat{\mathfrak B}}(R^{||},\{G\}C)=Refunded$, which will not change the eventual synchronicity stipulation status of $\{G\}C$ so $\operatorname{stipStatus}^{es}(R^{||},\{G\}C)=\operatorname{stipStatus}^{es}(R_0^{||},\{G\}C)$. And we also defined $R^x=R_0^x$, so we can conclude that $\operatorname{stipStatus}^x(R^x,\{G\}C)=\operatorname{stipStatus}^{es}(R^{||},\{G\}C)$.

Again, other conditions hold trivially by inductive hypothesis, so we can claim that $R^x \sim_A R^{||}$.

Round-Based Execution

Because $\pi.status = \mathbf{Right}$, we know that $t < t_0 + 2\hat{\delta}$. But we also know, by the intermediate semantics rule for abort that $t \ge t_0 + \hat{\delta}$, so κ_0 is in refund phase, which is consistent with the new status $\pi_i.status = \mathbf{Stip-Refunded}(A_i)$ for each user A_i . Other conditions hold by inductive hypothesis.

Stipulation Slash. $R^{||} = R_0^{||} \xrightarrow{sslash(\kappa_0, \mathbb{B}, B)} \Gamma^{||}$ and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \wedge R_0^x \sim_A R_0^{||}$.

We do not extend the $BitML^x$ and instead keep $R^x = R_0^x$. Since this intermediate move doesn't change the frontiers or authorizations, Run Conformance and Coherence are trivial.

Round-Based Execution

Let $\langle C, \pi \rangle_{\kappa_0}^{\mathbb{B}} \in \Gamma^{||}$ be the stipulation contract for κ_0 . We know that according to the intermediate semantics, $\pi.status =$ **Stip-Right** and will change to $\pi'.status =$ **Stip-Slashed**(\underline{B}). By the inductive hypothesis of round-based execution, κ_0 is either in compensation phase $(t_0 \leq t < t + \hat{\delta})$ or in refund phase $(t_0 + \hat{\delta} \leq t < t + 2\hat{\delta})$. In both cases, we know by inductive hypothesis that $\underline{A} \notin \mathcal{R}_{\kappa}^{ss}(R^{||})$, so $slash(\kappa, \mathbb{B}, \underline{A})$ would be an invalid move in the intermediate semantics. Other conditions hold by inductive hypothesis.

Stipulation Compensation. $R^{||} = R_0^{||} \xrightarrow{scompensation(\kappa_0, \mathbb{B}, \mathbb{B})} \Gamma^{||}$ and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \land R_0^x \sim_A R_0^{||}$.

We are compensating with a revealed step secret on a blockchain that didn't use it to init. This can be either the honest user cleaning up an incomplete initialization or the malicious user hurting themselves. The started initialization, if any, is already in the run, so our only extension will be to add authorizations to future active contracts that we now consider complete when increasing the compensation history sets of those contracts.

We will choose the new $BitML^x$ run to be $R^x = R_0^x R_a^x$, where R_a^x consists of authorizations ($C : \kappa$) for every user C and contract κ such that $\kappa \in \mathsf{maxFrontier}(R^x)$ and $C \in \mathbb{AU}_{\kappa}^A(R^{||}) \setminus \mathbb{AU}_{\kappa}^A(R^{||})$.

Round-Based Execution

Let $\langle C, \pi \rangle_{\kappa_0}^{\mathbb{B}} \in \Gamma^{||}$ be the stipulation contract for κ_0 . We know that according to the intermediate semantics, $\pi.status = \mathbf{Stip\text{-}Slashed}(B)$. By the inductive hypothesis of round-based execution, κ_0 is either in compensation phase $(t_0 \le t < t + \hat{\delta})$ or in refund phase $(t_0 + \hat{\delta} \le t < t + 2\hat{\delta})$. The status after the move will be $\pi'.status = \mathbf{Stip\text{-}Compensation}(B)$. But we know by the inductive hypothesis that $B \ne A$. Other conditions hold by inductive hypothesis.

Run Conformance

We need to prove that R_a^x is a valid extension of R_0^x and it conforms to the strategy of the honest user. The intuition behind this is that we are only completing already started authorizations, so we know that the honest user intended to authorize.

For every authorization $(C:\kappa) \in R_a^x$, we know that the authorization is valid because we are limiting ourselves to contracts $\kappa \in \mathsf{maxFrontier}(R^x)$ and we know by the Frontier Configurations Lemma, that there is an active contract κ in the last configuration of R^x .

We are also saying that $C \in \mathbb{AU}_{\kappa}^{A}(R^{||})$, which implies there is at least one partial authorization $(C : \kappa, \mathbb{B}') \in R^{||}$. This means that κ has the form of a priority choice, which we know to also be true in R^{x} by the Code Symmetry invariant.

For the case where C = A, again, there is at least one partial authorization $(C : \kappa, \mathbb{B}') \in R^{||}$, which implies, by the definition of our honest strategy compilation, that $A : \kappa'$ is conforming to A's strategy.

Coherence

Compensations do not change the maximal frontiers.

For every contract $\kappa \in \mathsf{maxFrontier}(R^x)$, we can reason that:

$$\begin{split} \mathcal{A}_{\kappa}(R^{x}) &= \mathcal{A}_{\kappa}(R_{0}^{x}) \cup (\mathbb{A}\mathbb{U}_{\kappa'}^{\mathbf{A}}(R^{||}) \setminus \mathbb{A}\mathbb{U}_{\kappa}^{\mathbf{A}}(R_{0}^{||})) \\ &= \mathbb{A}\mathbb{U}_{\kappa}^{\mathbf{A}}(R_{0}^{||}) \cup (\mathcal{A}_{\kappa}(R_{0}^{||}) \cap \{\mathbf{A}\}) \cup (\mathbb{A}\mathbb{U}_{\kappa}^{\mathbf{A}}(R^{||}) \setminus \mathbb{A}\mathbb{U}_{\kappa}^{\mathbf{A}}(R_{0}^{||})) \\ &= \mathbb{A}\mathbb{U}_{\kappa}^{\mathbf{A}}(R^{||}) \cup (\mathcal{A}_{\kappa}(R_{0}^{||}) \cap \{\mathbf{A}\}) \end{split}$$

Top-level Withdraw. $R^{||} = R_0^{||} \xrightarrow{cwithdraw(\kappa,\mathbb{B})} \Gamma^{||}$ and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \wedge R_0^x \sim_A R_0^{||}$. We start by proving round-based execution of the intermediate run.

We are doing a $cwithdraw(\kappa,\mathbb{B})$ in the intermediate semantics, that will consume an active contract $\langle \text{withdraw } \vec{\mathbf{B}} \to \vec{A}, \pi \rangle_{\kappa}^{\mathbb{B}}$ with $\pi.status = \textbf{Choice}$ and result in left-descendant terminal contracts $\langle C, \pi_1 \rangle_{\kappa_1}^{\mathbb{B}}, \dots, \langle C, \pi_n \rangle_{\kappa_n}^{\mathbb{B}}$, where for each withdrawing user $A_i \in \vec{A}$, $\pi_i.status = \textbf{Assigned}(A_i)$ and $\kappa_i = \kappa | L_i$.

Timeout Consistency

Trivial, as all successor contracts are terminal.

Round-Based Execution

Because $\pi.status =$ Choice, we know by inductive hypothesis that the round status of κ , roundStatus $(\kappa, R^{||}) = (r, p)$, can be either waiting to synchronise $(|\kappa| > r)$ or in compensation phase $(|\kappa| > r \land p = 0)$. We will split by cases on the possible round status of κ .

- If $|\kappa| > r$, then we need to prove that if there is another active contract $\langle \text{withdraw } \vec{B} \to \vec{A}, \pi \rangle_{\kappa}^{\hat{\mathbb{B}}}$ for some other blockchain $\hat{\mathbb{B}}$, then there are no right-descendants. This is the case, because for all descendants, $\kappa_i = \kappa |L_i$.
- If $|\kappa| > r \land p = 0$, then we know by the transition rule that for some user B, we have a revealed step secret, that is, $B \in \mathcal{R}^{ss}_{\kappa}(R^{||})$ and because $\pi.status = \mathbf{Choice}$, we know that $B \neq A$.

For every successor κ_i , we know that if they have a round status roundStatus $(\kappa_i, R^{||}) = (r_i, p_i)$. $|\kappa_i| > |\kappa| \ge r$. That is, the successor contracts are waiting to synchronize, which is consistent with being in an assigned state.

For proving Run Conformance and Coherence, we will split by cases on whether the $BitML^x$ contract κ is in the maximal frontier of R_0^x or not, i.e. on whether $\kappa \in \mathsf{maxFrontier}(R_0^x)$.

• If $\kappa \in \mathsf{maxFrontier}(R_0^x)$ then we are in the case where this is the first partial move for κ , which will extend the maximal frontier. We will extend the BitML^x run to mirror this movement, so we choose $R^x = R_0^x \xrightarrow{\mathit{cwithdraw}(\kappa)} \Gamma^x$ for this case.

Run Conformance

We want to prove that $\Sigma_A^x \vdash R^x$. Because we are doing a $cwithdraw(\kappa,\mathbb{B})$ in the intermediate semantics, we know that we have a contract $\langle C,\pi\rangle_\kappa^\mathbb{B} \in \Gamma_{R_0^{||}}$ with C= withdraw $\vec{\mathbb{B}}\to \vec{A}$. And because $\kappa\in$ maxFrontier(R_0^x), by the Frontier Configuration Lemma, we also know that for some balance B and some contract C', we have $\langle C',\mathbb{B}\rangle_\kappa\in\Gamma_{R^x}$. But we know by the Configuration Consistency invariant that C=C'.

Then, according to the $BitML^x$ semantics, $R^x \xrightarrow{cwithdraw(\kappa)}$ is a valid move. And because it does not depend on A's strategy, we can claim that $\Sigma_A^x \vdash R^x$.

Coherence

We need to prove that after extending both runs, we still have correspondence of maximal frontiers, that is, $\max Frontier(R^x) = \bigsqcup_{\mathbb{B}' \in \mathcal{B}} \max Frontier(R^{||}, \mathbb{B}')$. Intuitively, this should hold because we are extending the $BitML^x$ frontier and eventual synchronicity frontier in the same way.

We know that $\kappa \in \mathsf{maxFrontier}(R_0^x)$ and we are extending the run with a $\mathit{cwithdraw}(\kappa)$ move. If we consider the successors $\vec{\kappa}$ of κ after this move, we can see that the new maximal frontier of the BitML^x run will be

$$\mathsf{maxFrontier}(R^x) = \mathsf{maxFrontier}(R^x_0) \setminus \{\kappa\} \cup \{\kappa' \in \vec{\kappa}\} = \bigsqcup_{\mathbb{B}' \in \mathcal{B}} \mathsf{maxFrontier}(R^{||}_0, \mathbb{B}') \setminus \{\kappa\} \cup \{\kappa' \in \vec{\kappa}\}$$

And, because κ is in the eventual synchronicity frontier of $R_0^{||}$, we know by the Eventual Synchronicity Update Lemma that

$$\bigsqcup_{\mathbb{B}' \in \mathcal{B}} \mathsf{maxFrontier}(R^{||}, \mathbb{B}') = \bigsqcup_{\mathbb{B}' \in \mathcal{B}} \mathsf{maxFrontier}(R^{||}_0, \mathbb{B}') \setminus \{\kappa\} \cup \{\kappa' \in \vec{\kappa}\}$$

In conclusion, we proved that $\max Frontier(R^x) = \max Frontier(R^x)$

Configuration Consistency

The $BitML^x$ and intermediate semantics rules will introduce the same successor contracts and distribute the funds in the same way, so the invariant holds.

• If $\kappa \notin \mathsf{maxFrontier}(R_0^x)$ then we are in the case where this is not the first partial move, but we are instead replicating a move started by another blockchain and that we expect to already be present in the BitML^x run. For this reason, we will not extend the BitML^x run but instead keep $R^x = R_0^x$. We know by inductive hypothesis that this run is valid and conforming to the user strategy.

Coherence

We didn't change the $BitML^x$ run, so $\max Frontier(R^x) = \max Frontier(R^x_0)$. And by inductive hypothesis we know that $\max Frontier(R^x_0) = \bigsqcup_{\mathbb{B}' \in \mathcal{B}} \max Frontier(R^{\parallel}_0, \mathbb{B}')$. Additionally, because $\kappa \notin \max Frontier(R^x_0)$, we know that $\bigsqcup_{\mathbb{B}' \in \mathcal{B}} \max Frontier(R^{\parallel}_0, \mathbb{B}') = \bigsqcup_{\mathbb{B}' \in \mathcal{B}} \max Frontier(R^{\parallel}_0, \mathbb{B}')$. That is, this move didn't change the join of maximal intermediate frontiers. Intuitively, this is because we are replicating a similar move on another blockchain, so the join of maximal frontiers already has descendants of κ . This means that even if this move expands the maximal frontier of \mathbb{B} , the join will still be greater.

In conclusion, we proved that $\max Frontier(R^x) = \max Frontier(R^x)$

Configuration Consistency

Holds trivially, as $\kappa \notin \max Frontier(R_0^x)$.

Step secret reveal. $R^{||} = R_0^{||} \xrightarrow{B:s_{R||}^{\kappa}} \Gamma^{||}$ and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \land R_0^x \sim_A R_0^{||}$.

We do not extend the $BitML^x$ and instead keep $R^x = R_0^x$. Since this intermediate move doesn't change the frontiers or authorizations, Run Conformance and Coherence are trivial.

Round-Based Execution

We need to prove that the step secret reveal is not violating the invariants for the honest user.

Let $\langle D \Leftrightarrow C, \pi \rangle_{\mathcal{B}}^{\mathbb{B}} \in \Gamma_{R^{||}}$ be the active contract for κ and $(r,p) = \mathsf{roundStatus}(\kappa, R^{||})$. If B = A, we know according to the honest user strategy compilation that $\pi.status = \mathsf{Choice}$ and $t < \pi.time = t_0 + 2|\kappa|\hat{\delta}$. Solving for $|\kappa|$, we get that $|\kappa| > \frac{t-t_0}{2\delta} \ge \left\lfloor \frac{t-t_0}{2\delta} \right\rfloor = r$. That is, κ is waiting to synchronise, which means that there is no restriction in round-based execution for revealing the step secret.

Introduction of left. $R^{||} = R_0^{||} \xrightarrow{left(\kappa,\mathbb{B})} \Gamma^{||}$ and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \wedge R_0^x \sim_A R_0^{||}$.

We are doing a $left(\kappa, \mathbb{B})$ move in the intermediate semantics, that will change the status of an active contract $\langle \vec{A} \colon D \Leftrightarrow C, \pi \rangle_{\kappa}^{\mathbb{B}}$ from $\pi.status =$ **Choice** to $\pi'.status =$ **Left** and consume authorizations

We do not extend the $BitML^x$ and instead keep $R^x = R_0^x$. Since this intermediate move doesn't change the frontiers or authorizations, Run Conformance and Coherence are trivial.

Round-Based Execution

Let $\langle D \Leftrightarrow C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}$ be the active contract for κ . We know that according to the intermediate semantics, $\pi.status =$ **Choice** and by the inductive hypothesis of round-based execution, κ is either waiting to synchronize or in compensation phase. In both cases, the change to $\pi'.status = \mathbf{Left}$ is a valid one.

In the case where κ is in compensation phase, we know by the intermediate semantics that there is a step secret s_{κ}^{B} revealed for some user B and by the inductive hypothesis that $B \neq A$.

Authorized Left

We are introducing a contract with **Left** status, so we need to prove that if A is one of the users authorizing the contract and we have the active contract in the $BitML^x$ configuration, then we also have A's authorization for it.

If $\langle \vec{A} \colon D \Leftrightarrow C, \mathbf{B} \rangle_{\kappa} \in \Gamma_{R^x}$, then $\kappa \in \mathsf{maxFrontier}(R^x)$. We know that $A \in \mathcal{A}_{\kappa}(R^{||})$, so by Coherence we also know that $A \in \mathcal{A}_{\kappa}(R^x)$. Then by the Authorizations Persistance lemma, we know that $A[\kappa \triangleright D] \in \Gamma_{R^x}$.

The proof for the round-based execution invariant is analogous to the *cwithdraw* case and for timeout consistency it is also trivial.

For Run Conformance, Coherence and Configuration Consistency, we will again start by splitting by cases on whether the $BitML^x$ contract κ is in the maximal frontier of R_0^x or not, i.e. on whether $\kappa \in \mathsf{maxFrontier}(R_0^x)$.

• If $\kappa \in \mathsf{maxFrontier}(R_0^x)$ then we are in the case where this is the first partial move for κ , which will extend the maximum frontier. We will extend the BitML^x run to mirror this movement, like we did on the $\mathit{cwithdraw}$ case. But additionally, we might need to add missing authorizations if needed.

We will choose $R^x = R_0^x R_a^x \xrightarrow{dwithdraw(\kappa)} \Gamma^x$, where R_a^x is a sequence of $\xrightarrow{B:\kappa}$ transitions for every $B \in \vec{B}: B[\kappa \triangleright \text{withdraw } \vec{B} \rightarrow \vec{A}] \notin \Gamma_{R^x}$. That is, we first extend with all missing authorizations needed and then with the withdraw move.

Run Conformance

We know by the Configuration Consistency invariant that $\langle C, \mathbf{B} \rangle_{\kappa} \in \Gamma_{R^x}$. This makes every individual authorization move $(\mathbf{B}:\kappa) \in R_a^x$ a valid transition, and because their only change to the configuration is adding the authorization object, we can conclude that R_a^x is a valid extension. It is also conformant to the honest user strategy because we know by the Honest Left invariant that authorization from \mathbf{A} already. That is, $\mathbf{A}[\kappa \triangleright \text{withdraw } \vec{\mathbf{B}} \to \vec{\mathbf{A}}] \in \Gamma_{R_a^{||}}$.

And because after R_a^x we have all needed authorizations, $dwithdraw(\kappa)$ is a valid move in $R_0^x R_a^x$. It can be scheduled independently of the honest user, so $\Sigma_A^x \vdash R^x$.

Coherence

The reasoning for the successor assigned contracts is analogous to that in the cwithdraw case.

But because we are possibly adding authorizations, we need to prove that we hold the $\mathcal{A}_{\kappa}(R^x) = \mathbb{A}\mathbb{U}^{\mathbf{A}}_{\kappa}(R^{||}) \cup (\mathcal{A}_{\kappa}(R^{||}) \cap \{\mathbf{A}\})$ invariant. But this is very straight forward, because the $dwithdraw(\kappa)$ move in the $BitML^x$ run will consume the active contract for κ , so $\kappa \notin \max Frontier(R^x)$.

Configuration Consistency

Analogous to the case of *cwithdraw*.

• If $\kappa \notin \mathsf{maxFrontier}(R_0^x)$ then we are in the case where this is not the first partial move, but we are instead replicating a move started by another blockchain and that we expect to already be present in the BitML^x run. For this reason, we will not extend the BitML^x run but instead keep $R^x = R_0^x$. We know by inductive hypothesis that this run is valid and conforming to the user strategy. The proofs for Coherence and Configuration Consistency are analogous to the case of $\mathit{cwithdraw}$.

Proofs for Run Conformance Coherence and Configuration Consistency are analogous to the dwithdraw case.

Timeout Consistency

Because $\pi.status = \textbf{Left}$, we know by inductive hypothesis that $\pi.time = t_0 + 2|\kappa|\hat{\delta}$ where t_0 is the stipulation time of κ 's root contract. Then, for every successor κ_i , we can reason in the following way:

$$\pi_i.time = \pi.time + 2\hat{\delta} = t_0 + 2|\kappa|\hat{\delta} + 2\hat{\delta} = t_0 + 2(|\kappa| + 1)\hat{\delta} = t_0 + 2(|\kappa_i|)\hat{\delta}$$

Round-Based Execution

Because $\pi.status = \textbf{Left}$, we know by inductive hypothesis that the round status roundStatus $(\kappa, R^{||}) = (r, p)$ of κ can be either waiting to synchronise $(|\kappa| > r)$ or in compensation phase $(|\kappa| > r \land p = 0)$. We will split by cases on the possible round status of κ .

- If $|\kappa| > r$, then we need to prove that if there is another active contract $\langle \text{split } \vec{B} \to \vec{C}, \pi \rangle_{\kappa}^{\hat{\mathbb{B}}}$ for some other blockchain $\hat{\mathbb{B}}$, then there are no right-descendants. This is the case, because for all descendants, $\kappa_i = \kappa |L_i$.
- If $|\kappa| > r \wedge p = 0$, because κ has succesor contracts in $R^{||}$, we need to prove that $\mathcal{R}^{ss}_{\kappa_0}(R^{||}) \setminus \{A\} \neq \emptyset$, but we already know this by inductive hypothesis because $\pi.status = \mathbf{Left}$.

For every successor κ_i , we know that $|\kappa_i| > |\kappa| \ge r$. That is, the successor contracts are waiting to synchronize, which is consistent with being in a choice state.

Reveal. $R^{||} = R_0^{||} \xrightarrow{reveal(\kappa, \mathbb{B})} \Gamma^{||}$ and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \land R_0^x \sim_A R_0^{||}$.

We are doing a $reveal(\kappa, \mathbb{B})$ move in the intermediate semantics, that will consume an active contract $\langle \vec{B} \rangle$: reveal \vec{s} if p then $C \Leftrightarrow C', \pi \rangle_{\kappa}^{\mathbb{B}}$ (where \vec{B} and \vec{s} are possibly empty), $\pi.status = \mathbf{Choice}$ and result in a left-descendant active contract $\langle C, \pi' \rangle_{\kappa'}^{\mathbb{B}}$, with $\pi'.status = \mathbf{Choice}$ and $\kappa' = \kappa | L_0$.

For Run Conformance and Coherence we again split on whether $\kappa \in \max Frontier(\mathbb{R}^x)$.

• If $\kappa \in \mathsf{maxFrontier}(R_0^x)$ then we are in the case where this is the first partial move for κ , which will extend the maximum frontier. We will extend the BitML^x run to mirror this movement and, like in the $\mathit{dwithdraw}$ case, we will add all of the missing authorizations first.

We will choose $R^x = R_0^x R_a^x \xrightarrow{reveal(\kappa)} \Gamma^x$, where R_a^x is a sequence of $\xrightarrow{\pmb{B}:\kappa}$ transitions for every $\pmb{B} \in \vec{\pmb{B}}: \pmb{B}[\kappa \triangleright \text{reveal } \vec{s} \text{ if } p \text{ then } C] \notin \Gamma_{R^x}$.

Run Conformance

We know by the Configuration Consistency invariant that $\langle \vec{B} \colon \texttt{reveal} \ \vec{s} \ \texttt{if} \ p \ \texttt{then} \ C \Leftrightarrow C', \mathbf{B} \rangle_{\kappa} \in \Gamma_{R^x}$. This makes every individual authorization move $(B \colon \kappa) \in R^x_a$ a valid transition, and because their only change to the configuration is adding the authorization object, we can conclude that R^x_a is a valid extension.

We know by the intermediate semantics transition rule for reveal that $A[(\kappa,\mathbb{B}) \triangleright D] \in \Gamma_{R_0^{||}}$. This is only possible if $(A:\kappa,\mathbb{B}) \in R_0^{||}$ so by Coherence we also know that $(A:\kappa)R_0^{||}$. Then, by the Authorization Persistance lemma, we know that $A[\kappa \triangleright \texttt{reveal} \ \vec{s} \ \texttt{if} \ p \ \texttt{then} \ C] \in \Gamma_{R^x}$, meaning that we will not be authorizing for A in R_a^x , so the extension is conformant to the honest user strategy.

We also know by the intermediate semantics transition rule for reveal that $\{s \in \vec{s}\} \subseteq \mathcal{R}^u(R_0^{||})$ and Coherence we know that $\mathcal{R}^u(R_0^{||}) = \mathcal{R}^u(R_0^x)$. Additionally we also know by the intermediate semantics that $[p]_{secrets} = true$. And because after R_a^x we have all needed authorizations, $reveal(\kappa)$ is a valid move in $R_0^x R_a^x$. It can be scheduled independently of the honest user, so $\Sigma_A^x \vdash R^x$.

Proofs for Coherence and Configuration Consistency are analogous to the dwithdraw case.

• If $\kappa \notin \mathsf{maxFrontier}(R_0^x)$ then we don't extend the BitML^x , keeping instead $R^x = R_0^x$. Run conformance is trivial by inductive hypothesis and Coherence is analogous to the $\mathit{dwithdraw}$ case.

The proofs for Round-based Execution and Timeout Consistency are analogous to the case of *split*, for the single successor contract.

Right. $R^{||} = R_0^{||} \xrightarrow{right(\kappa, \mathbb{B})} \Gamma^{||}$ and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \wedge R_0^x \sim_A R_0^{||}$.

We do not extend the $BitML^x$ and instead keep $R^x = R_0^x$. Since this intermediate move doesn't change the frontiers or authorizations, Run Conformance, Coherence and Configuration Consistency are trivial.

Timeout Consistency

Because $\pi.status =$ Choice, we know by inductive hypothesis that $\pi.time = t_0 + 2|\kappa|\hat{\delta}$ where t_0 is the stipulation time of κ 's root contract. Then, for the new status π' after the move, we can reason in the following way:

$$\pi'.time = \pi.time + \hat{\delta} = t_0 + 2|\kappa|\hat{\delta} + \hat{\delta} = t_0 + (2|\kappa| + 1)\hat{\delta}$$

Round-Based Execution

Let $\langle D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma^{||}$ be the active contract for κ . We know that according to the intermediate semantics, $\pi.status = \mathbf{Choice}$. By the inductive hypothesis of round-based execution, κ is either waiting to synchronize $(|\kappa| < r)$ or in compensation phase $(|\kappa| = r \land p = 0)$. But we also know that $t \ge \pi.time = t_0 + 2|\kappa|\hat{\delta}$. Solving for $|\kappa|$, we get that $|\kappa| \le \frac{t - t_0}{2\hat{\delta}}$ which implies that $r = \left\lfloor \frac{t - t_0}{2\hat{\delta}} \right\rfloor \ge |\kappa|$. That means κ is in compensation phase. Other conditions hold by inductive hypothesis.

Skip. $R^{||} = R_0^{||} \xrightarrow{skip(\kappa,\mathbb{B})} \Gamma^{||}$ and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \wedge R_0^x \sim_A R_0^{||}$.

We are doing a $skip(\kappa, \mathbb{B})$ move in the intermediate semantics, that will consume an active contract $\langle D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbb{B}}$ with $\pi.status = \mathbf{Right}$ and result in a single right-descendant active contract $\langle C, \pi^{\downarrow} \rangle_{\kappa^{\downarrow}}^{\mathbb{B}}$, where, $\pi^{\downarrow}.status = \mathbf{Choice}$ and $\kappa^{\downarrow} = \kappa | R \in rdesc(\kappa)$.

For proving Run Conformance, Coherence and Configuration Consistency, we will split by cases on whether the $BitML^x$ contract κ is in the maximal frontier of R_0^x or not, i.e. on whether $\kappa \in \mathsf{maxFrontier}(R_0^x)$.

• If $\kappa \in \mathsf{maxFrontier}(R_0^x)$ then we are in the case where this is the first partial move for κ , which will extend the maximal frontier. We will extend the BitML^x run to mirror this movement, so we choose $R^x = R_0^x \xrightarrow{skip(\kappa, \vec{\kappa})} \Gamma^x$ for this case.

Run Conformance

We want to prove that $\Sigma_A^x \vdash R^x$. We know by the Configuration Consistency invariant that we have an active contract $\langle D \Leftrightarrow C, \mathbf{B} \rangle_{\kappa} \in \Gamma_{R^{||}}$, but the $BitML^x$ semantics state that the move needs to be done by consensus. That is, it needs to be in the output of the strategies for all users. In particular, we need to prove that $skip(\kappa) \in \Sigma_A^x(R_0^{||})$. But, because $\pi.status = \mathbf{Right}$, we know this to be true by the honest skip invariant.

Coherence

Proofs for Coherence and Configuration Consistency are analogous to the dwithdraw case.

• If $\kappa \notin \mathsf{maxFrontier}(R_0^x)$, then again, we will not extend the BitML^x run but instead keep $R^x = R_0^x$. We know by inductive hypothesis that this run is valid and conforming to the user strategy. The proofs for Coherence and Configuration Consistency are analogous to the case of $\mathit{cwithdraw}$.

Timeout Consistency

Because $\pi.status = \textbf{Right}$, we know by inductive hypothesis that $\pi.time = t_0 + (2|\kappa| + 1)\hat{\delta}$ where t_0 is the stipulation time of κ 's root contract. Then, the successor κ' , we can reason in the following way:

$$\pi'.time = \pi.time + \hat{\delta} = t_0 + (2|\kappa| + 1)\hat{\delta} + \hat{\delta} = t_0 + (2|\kappa| + 2)\hat{\delta} = t_0 + 2(|\kappa| + 1)\hat{\delta} = t_0 + 2(|\kappa'|)\hat{\delta}$$

Round-Based Execution

Because $\pi.status = \textbf{Right}$, we know by inductive hypothesis that the round status roundStatus $(\kappa, R^{||}) = (r, p)$ of κ can be either compensation phase $(|\kappa| > r \land p = 0)$ or skipping phase $(|\kappa| > r \land p = 1)$. But we also know by the transition rule for skip that $t \ge \pi.time = t_0 + (2|\kappa| + 1)\hat{\delta}$ which implies that p = 1.

Because we have a right-descendant for κ , we need to prove that if there are other contracts $\langle D \Leftrightarrow C, \pi \rangle_{\kappa}^{\mathbb{B}}$, then there are no left descendants. We will prove this by contradiction. Suppose the opposite was true and $ldesc(\kappa) \neq \emptyset$. That would imply by inductive hypothesis that $\exists B \in \mathcal{U}_{\kappa}(R^{||}) \setminus \{A\} : \pi.status = \mathbf{CompensatedFrom}(B)$. Which is a contradiction because $\pi.status = \mathbf{Right}$.

For the successor κ^{\downarrow} , we know that $|\kappa^{\downarrow}| > |\kappa| \ge r$. That is, the successor contract is waiting to synchronize, which is consistent with being in a choice state.

Authorizations. $R^{||} = R_0^{||} \xrightarrow{B: \kappa, \mathbb{B}} \Gamma^{||}$ and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \land R_0^x \sim_A R_0^{||}$. We need to distinguish a few different cases here. Firstly, whether the authorization is for a contract on the maximal frontier.

We need to distinguish a few different cases here. Firstly, whether the authorization is for a contract on the maximal frontier. If $\kappa \notin \mathsf{maxFrontier}(R_0^x)$, then we can ignore this transition and keep $R^x = R_0^x$ which is trivially conforming and (given that the maximal frontier doesn't change) coherent.

If $\kappa \in \mathsf{maxFrontier}(R_0^x)$, the next distinction on whether the user authorizing is our honest user A or some other potentially malicious user. That is, whether B = A. The general strategy will be that for the case where it's the honest user authorizing we will mirror the authorization in the $BitML^x$ run when observing the authorization on the first blockchain, while for other users we will take a more conservative approach and only mirror the authorization on the last blockchain.

- If B = A, we need to further distinguish whether this is the first blockchain where the honest user is authorizing the move or not.
- If $\underline{A} \notin \mathcal{A}_{\kappa}(R_0^{||})$ then we will mirror this authorization in the $BitML^x$ run. That is, we will choose $R^x = R_0^x \xrightarrow{A: \kappa} \Gamma^{||}$.

Conformance

We first need to prove that A: κ is a valid move in R_0^x . For this, the $BitML^x$ semantics requires that we have an active priority choice contract κ in the latest configuration of R^x . Because we are doing a A: κ , $\mathbb B$ transition, the intermediate semantics requires that there is an active contract $\langle D \Leftrightarrow C, \pi \rangle_{\kappa}^{\mathbb B}$ in $\Gamma_{R||}$. By the Frontier Configuration Lemma, we also know that for some balance B and some contract C', we have $\langle C', B \rangle_{\kappa} \in \Gamma_{R^x}$. But we know by the code symmetry invariant that C = C'.

Additionally, we know that $\Sigma_A^{||} \models R^{||}$, so it must be the case that this authorization is on the in intermediate semantics strategy of A. Then, by the definition of our honest user strategy compilation, we also know that authorizing is on A's $BitML^x$ strategy.

Coherence

We need to prove that $\mathcal{A}_{\kappa}(R^x) = \mathbb{AU}_{\kappa}^{\mathbf{A}}(R^{||}) \cup (\mathcal{A}_{\kappa}(R^{||}) \cap \{\mathbf{A}\})$. If we observe definitions of R^x and $R^{||}$, we can see that $\mathcal{A}_{\kappa}(R^x) = \mathcal{A}_{\kappa}(R_0^x) \cup \{\mathbf{A}\}$ and $\mathcal{A}_{\kappa}(R^{||}) = \mathcal{A}_{\kappa}(R_0^{||}) \cup \{\mathbf{A}\}$. Then, we have:

$$\begin{split} \mathcal{A}_{\kappa}(R^{x}) &= \mathcal{A}_{\kappa}(R_{0}^{x}) \cup \{A\} \\ &= \mathbb{A}\mathbb{U}_{\kappa}^{A}(R_{0}^{||}) \cup (\mathcal{A}_{\kappa}(R_{0}^{||}) \cap \{A\}) \cup \{A\} \\ &= \mathbb{A}\mathbb{U}_{\kappa}^{A}(R_{0}^{||}) \cup ((\mathcal{A}_{\kappa}(R_{0}^{||}) \cup \{A\}) \cap \{A\}) \\ &= \mathbb{A}\mathbb{U}_{\kappa}^{A}(R_{0}^{||}) \cup (\mathcal{A}_{\kappa}(R^{||}) \cap \{A\}) \end{split}$$

- If $\underline{A} \in \mathcal{A}_{\kappa}(R_0^{||})$, then we don't extend the run and keep $R^x = R_0^x$, which is trivially conforming. To see that it's also coherent, we can do:

$$\begin{split} \mathcal{A}_{\kappa}(R^{x}) &= \mathcal{A}_{\kappa}(R_{0}^{x}) \\ &= \mathbb{A}\mathbb{U}_{\kappa}^{A}(R_{0}^{\parallel}) \cup (\mathcal{A}_{\kappa}(R_{0}^{\parallel}) \cap \{A\}) \\ &= \mathbb{A}\mathbb{U}_{\kappa}^{A}(R_{0}^{\parallel}) \cup ((\mathcal{A}_{\kappa}(R_{0}^{\parallel}) \cup \{A\}) \cap \{A\}) \\ &= \mathbb{A}\mathbb{U}_{\kappa}^{A}(R_{0}^{\parallel}) \cup (\mathcal{A}_{\kappa}(R^{\parallel}) \cap \{A\}) \end{split}$$

- If $B \neq A$, we need to further distinguish whether this is the last blockchain where the potentially malicious user is authorizing the move or not, up to compensations for A.
 - If $B \in \mathbb{AU}_{\kappa}^{A}(R^{||})$, then we extend the $BitML^{x}$ run with the corresponding authorization. We will choose $R^{x} = R_{0}^{x} \xrightarrow{\stackrel{R}{B}: \kappa} \Gamma^{||}.$

Conformance

Idem for the case of A, except that this time we don't care about conforming to the strategy, because it's not an honest user.

Coherence

We need to prove that $\mathcal{A}_{\kappa}(R^x) = \mathbb{AU}_{\kappa}^{A}(R^{||}) \cup (\mathcal{A}_{\kappa}(R^{||}) \cap \{A\})$. We start by observing that B is finishing the authorization in this action, so $\mathbb{AU}_{\kappa}^{A}(R^{||}) = \{B\} \cup \mathbb{AU}_{\kappa}^{A}(R_{0}^{||})$. Then, we have:

$$\mathcal{A}_{\kappa}(R^{x}) = \{ \mathbf{B} \} \cup \mathcal{A}_{\kappa}(R_{0}^{x})$$

$$= \{ \mathbf{B} \} \cup \mathbb{AU}_{\kappa}^{\mathbf{A}}(R_{0}^{||}) \cup (\mathcal{A}_{\kappa}(R_{0}^{||}) \cap \{ \mathbf{A} \})$$

$$= \mathbb{AU}_{\kappa}^{\mathbf{A}}(R^{||}) \cup (\mathcal{A}_{\kappa}(R^{||}) \cap \{ \mathbf{A} \})$$

- If $\underline{B} \notin \mathbb{AU}_{\kappa}^{\underline{A}}(R^{||})$, then we don't extend the run and keep $R^x = R_0^x$, which is trivially conforming. To see that it's also coherent, we can do:

$$\begin{split} \mathcal{A}_{\kappa}(R^{x}) &= \mathcal{A}_{\kappa}(R_{0}^{x}) \\ &= \mathbb{A}\mathbb{U}_{\kappa}^{A}(R_{0}^{||}) \cup (\mathcal{A}_{\kappa}(R_{0}^{||}) \cap \{A\}) \\ &= \mathbb{A}\mathbb{U}_{\kappa}^{A}(R^{||}) \cup (\mathcal{A}_{\kappa}(R^{||}) \cap \{A\}) \end{split}$$

authorizations, Run Conformance and Coherence are trivial.

Round-Based Execution

Let $\langle D \Leftrightarrow C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma^{||}$ be the active contract for κ . We know that according to the intermediate semantics, $\pi.status = \mathbf{Right}$ and will change to $\pi'.status = \mathbf{Slashed}(B)$. By the inductive hypothesis of round-based execution, κ is either in compensation phase $(|\kappa| = r \land p = 0)$ or in skipping phase $(|\kappa| = r \land p = 1)$. In both cases, we know by inductive hypothesis that $A \notin \mathcal{R}_{\kappa}^{ss}(R^{||})$, so $slash(\kappa, \mathbb{B}, A)$ would be an invalid move in the intermediate semantics. Other conditions hold by inductive hypothesis.

 $\textbf{Compensations.} \ \ R^{||} = R_0^{||} \xrightarrow{compensate(\kappa, \mathbb{B}, \textbf{\textit{B}})} \Gamma^{||} \ \ \text{and we know by inductive hypothesis that} \ \ \Sigma_{\textbf{\textit{A}}}^x \vdash R_0^x \land R_0^x \sim_{\textbf{\textit{A}}} R_0^{||}.$

We are compensating with a revealed step secret on a blockchain that didn't use it to perform a left move. This can be either the honest user cleaning up an incomplete move or the malicious user hurting themselves. The started move, if any, is already in the run, so our only extension will be to add authorizations that we now consider complete when increasing the compensation history sets of those contracts.

Run Conformance

We will choose the new $BitML^x$ run to be $R^x = R_0^x R_a^x$, where R_a^x consists of authorizations ($C: \kappa'$) for every user Cand contract κ' such that $\kappa' \in \mathsf{maxFrontier}(R^x)$ and $C \in \mathbb{AU}_{\kappa'}^{C}(R^{||}) \setminus \mathbb{AU}_{\kappa'}^{A}(R_0^{||})$.

We need to prove that R_a^x is a valid extension of R_0^x and it conforms to the strategy of the honest user. The intuition behind this is that we are only completing already started authorizations, so we know that the honest user intended to authorize.

For every authorization $(C: \kappa') \in R_a^x$, we know that the authorization is valid because we are limiting ourselves to contracts $\kappa' \in \mathsf{maxFrontier}(R^x)$ and we know by the Frontier Configurations Lemma, that there is an active contract κ' in the last configuration of R^x .

We are also saying that $C \in \mathbb{AU}_{\kappa'}^{A}(R^{||})$, which implies there is at least one partial authorization $(C : \kappa', \mathbb{B}') \in R^{||}$. This means that κ' has the form of a priority choice, which we know to also be true in R^x by the Configuration Consistency invariant

For the case where C = A, again, there is at least one partial authorization $(C : \kappa', \mathbb{B}') \in R^{||}$, which implies, by the definition of our honest strategy compilation, that $A : \kappa'$ is conforming to A's strategy.

Coherence

Compensations do not change the intermediate frontiers, so in particular they don't change the maximal frontier. If we look at the authorizations for κ' , we can reason that:

$$\begin{split} \mathcal{A}_{\kappa}(\boldsymbol{R}^{x}) &= \mathcal{A}_{\kappa}(\boldsymbol{R}_{0}^{x}) \cup (\mathbb{A}\mathbb{U}_{\kappa'}^{\mathbf{A}}(\boldsymbol{R}^{||}) \setminus \mathbb{A}\mathbb{U}_{\kappa'}^{\mathbf{A}}(\boldsymbol{R}_{0}^{||})) \\ &= \mathbb{A}\mathbb{U}_{\kappa}^{\mathbf{A}}(\boldsymbol{R}_{0}^{||}) \cup (\mathcal{A}_{\kappa}(\boldsymbol{R}_{0}^{||}) \cap \{\boldsymbol{A}\}) \cup (\mathbb{A}\mathbb{U}_{\kappa'}^{\mathbf{A}}(\boldsymbol{R}^{||}) \setminus \mathbb{A}\mathbb{U}_{\kappa'}^{\mathbf{A}}(\boldsymbol{R}_{0}^{||})) \\ &= \mathbb{A}\mathbb{U}_{\kappa'}^{\mathbf{A}}(\boldsymbol{R}^{||}) \cup (\mathcal{A}_{\kappa}(\boldsymbol{R}_{0}^{||}) \cap \{\boldsymbol{A}\}) \end{split}$$

Round-Based Execution

We are doing a $compensation(\kappa, \mathbb{B}, B)$ move in the intermediate run. We know by the intermediate semantics that we have an active contract $\langle C, \pi \rangle_{\kappa}^{\mathbb{B}}$ with $\pi.status = \mathbf{Slashed}(B)$, and by inductive hypothesis that the round status roundStatus $(\kappa, R^{||}) = (r, p)$ of κ can be either compensation phase $(|\kappa| > r \land p = 0)$ or skipping phase $(|\kappa| > r \land p = 1)$. The status after the move will be $\pi'.status = \mathbf{CompensatedFrom}(B)$. But we know by the inductive hypothesis that $B \neq A$.

<u>Time Delay.</u> $R^{||} = R_0^{||} \xrightarrow{\delta} \Gamma^{||}$ and we know by inductive hypothesis that $\Sigma_A^x \vdash R_0^x \wedge R_0^x \sim_A R_0^{||}$. Let t be the time at the latest configuration of $R_0^{||}$. The time for $R^{||}$ after the delay move will be $t + \delta$.

The intermediate semantics states that a time delay needs to be agreed by all participants. That is, for all participants B, there should be a delay $\delta_B \in \Sigma_B^{\parallel}(R_0^{\parallel})$ such that $\delta_B \geq \delta$. In particular, for the honest user A, the strategy compilation specifies that they will only include a time delay when no condition for another action is met. Furthermore, the honest user strategy will always wait the minimum time until the next timeout is met for some active contract in R_0^{\parallel} , so we know that $\delta \leq \delta_A \leq \hat{\delta}$.

We do not extend the RiM^x run in this case and keep $R^x = R^x$. The delay doesn't change frontiers, so this run is trivially

We do not extend the $BitML^x$ run in this case and keep $R^x = R_0^x$. The delay doesn't change frontiers, so this run is trivially conformant, coherent configuration-consistent and timeout-consistent.

Round-Based Execution

Let $\{G\}^{\mathbb{B}}C \in \Gamma^{||}$ be a contract advertisement with stipulation time t_0 . We are interested in the case where $t < t_0 + \hat{\delta} \le t + \delta$, as for other cases, the invariant holds by inductive hypothesis. For said case, we can prove by contradiction $\operatorname{stipStatus}^{\mathbb{B}}(R_0^{||}, \kappa_0) = DoubleSpent$. Suppose that the opposite is true. That implies in particular that for every user $A_i \in users(G)$ there is a deposit $(A_i, v_i)_{x_i}^{\mathbb{B}} \in \Gamma_{R_i^{||}}$ fulfilling their deposit precondition $A_i : !B_i@\vec{x_i} \in G$.

But, if $t_0 + \hat{\delta} \leq t + \delta$ and given that we are in the case where $\delta \leq \hat{\delta}$, then even before the delay move, $t \geq t_0$. In particular for A, according to the honest user strategy compilation, this means that $doubleSpend(\{G\}^{\mathbb{B}}C, A) \in \Sigma_A^{||}(R_0^{||})$ and so $\Sigma_A^{||} \not\models R^{||}$.

We now move to prove the conditions over stipulation contracts. Let $\langle C, \pi \rangle_{\kappa_0}^{\mathbb{B}} \in \Gamma^{||}$ be a stipulation contract with stipulation time t_0 . We are interested in the cases where the round status of κ_0 changed with the delay, as for other cases, the conditions will hold by inductive hypothesis. Let's examine those cases:

• If $t < t_0 \le t + \delta$ then κ_0 is moving from initialization to compensation phase, so we need to prove that the adversary cannot slash the honest user. We know by inductive hypothesis that $\pi.status = \mathbf{Stip-Choice}$. We can prove by contradiction that $A \notin \mathcal{R}^{ss}_{\kappa_0}(R^{||})$. Assume that the opposite is true and A revealed their step secret. Then by inductive hypothesis we know that $\mathcal{R}^{is}_{\kappa_0}(R^{||}) = \mathcal{U}_{\kappa_0}(R^{||})$. According to the honest user strategy compilation rules, this means that $init(\kappa_0, \mathbb{B}) \in \Sigma_A^{||}$ and furthermore, $\delta \notin \Sigma_A^{||}$ which would imply $\Sigma_A^{||} \models R^{||}$.

- If t < t₀ + δ̂ ≤ t + δ then κ₀ is moving from compensation phase to refund phase. We know by inductive hypothesis that A ∉ R^{ss}_{κ₀}(R^{||}) and π.status ∉ {Stip-Slashed(A), Stip-Compensation(A)}.
 We can prove by contradiction that π.status ≠ Stip-Choice and ∀B ∈ U_{κ₀}(R^{||})π.status ≠ Slashed(B). If the opposite was true and the contract was in one of those statuses, then another move would be scheduled by the honest user strategy (skip(κ₀, B)) in the case of π.status = Stip-Choice or compensate(κ₀, B, B) in the case of π.status = Slashed(B). This would again mean that the honest user strategy would not wait and Σ^{||}_A ⊭ R^{||}.
- If $t < t_0 + 2\hat{\delta} \le t + \delta$, then κ_0 is moving from refund phase to finalized. We know by inductive hypothesis that $\pi.status = \mathbf{Stip}\mathbf{-Right} \lor \exists \mathbf{B} \in \mathcal{U}_{\kappa}(R^{||}) : \pi.status = \mathbf{Stip}\mathbf{-Refunded}(\mathbf{B}) \lor \exists \mathbf{B} \in \mathcal{U}_{\kappa}(R^{||}) \setminus \{\mathbf{A}\} : \pi.status \in \{\mathbf{Stip}\mathbf{-Slashed}(\mathbf{B}), \mathbf{Stip}\mathbf{-Compensation}(\mathbf{B})\}.$

We can prove by contradiction that $\pi.status \neq \mathbf{Stip}$ -Right. The proof is analogous to the previous case.

We now move to proving active contract conditions. Let $\kappa \in \Gamma^{||}$ be an active contract with round status roundStatus $(\kappa, R_0^{||}) = (r_0, p_0)$ before the delay move and roundStatus $(\kappa, R^{||}) = (r, p)$ after it. We split by cases on the relation between the status before and after the delay move:

- The cases where $r < r_0$ or $r = r_0 \land p < p_0$ are impossible, because they would imply a negative delay, which is not allowed by the intermediate semantics.
- If $r_0 \le r < |\kappa|$ or $|\kappa| < r_0 \le r$ or $r = r_0 = |\kappa| \land p = p_0$, then the invariant holds trivially by inductive hypothesis.
- The cases where $r_0 < |\kappa| \land r = |\kappa| \land p = 1$ or $r_0 < |\kappa| \land r > |\kappa|$ or $r_0 = |\kappa| \land p_0 = 0 \land r > |\kappa|$ are in contradiction with the honest user strategy, which will always wait the minimum time to meet the next timeout.
- If $r_0 < |\kappa| \land r = |\kappa| \land p = 0$, then κ is moving from waiting to synchronize into compensation phase. We know by inductive hypothesis that $\pi.status \in \{\text{Choice}, \text{Left}\} \lor \exists B \in \mathcal{U}_{\kappa}(R^{||}) : \pi.status = \text{Assigned}(B)$. If $\pi.status \in \{\text{Choice}, \text{Left}\}$, we can prove by contradiction that $A \notin \mathcal{R}^{ss}_{\kappa_0}(R^{||})$. Assume that the opposite is true and A revealed their step secret. According to the honest user strategy compilation rules, this means that, depending on the form of the contract, $ileft(\kappa_0, \mathbb{B}) \in \Sigma_A^{||}$ or $reveal(\kappa_0, \mathbb{B}) \in \Sigma_A^{||}$ and furthermore, $\delta \notin \Sigma_A^{||}$ which would imply $\Sigma_A^{||} \not \models R^{||}$. We know by inductive hypothesis that $desc(\kappa) \subseteq desc(\kappa)$. So if $\pi.status = \text{Left} \lor desc(\kappa) \neq \emptyset$, there must be an $ileft(\kappa, \hat{\mathbb{B}})$ or $reveal(\kappa, \hat{\mathbb{B}})$ move in $R_0^{||}$. According to the intermediate semantics, this implies that for some user B, we have a step secret $s_\kappa^B \in \Gamma_{R^{||}}$.
- If $r_0 = |\kappa| \wedge p_0 = 0 \wedge r = |\kappa| \wedge p = 1$, then κ is moving from compensation to skipping phase. We know by inductive hypothesis that $\pi.status \notin \{Slashed(A), CompensatedFrom(A)\}$. We can also prove by contradiction that $\pi.status \notin \{Choice, Left\}$. Assume that the contrary is true:
 - If $\pi.status = \textbf{Left}$, then we know by inductive hypothesis that $\mathcal{R}^{ss}_{\kappa_0}(R^{||}) \setminus \{A\} \neq \emptyset$ and there is a valid $\alpha(\kappa, \mathbb{B})$ move with $\alpha \in \{cwithdraw, dwithdraw, split\}$. According to the honest user strategy compilation, $\alpha(\kappa, \mathbb{B}) \in \Sigma_A^{||}(R^{||})$.
 - If $\pi.status = \textbf{Choice}$, we can observe that by the timeout consistency invariant $\pi.time = t_0 + 2|\kappa|\mathring{\delta}$, where t_0 is the stipulation time of the root contract of κ . By the definition of round-based status $r_0 = \left\lfloor \frac{t-t_0}{2\mathring{\delta}} \right\rfloor$. But we are in the case where $r_0 = |\kappa|$ so we can solve for t as $t \geq t_0 + 2\mathring{\delta}|\kappa|$. Combining these conditions, we get $t \geq \pi.time$. Then $skip(\kappa,\mathbb{B})$ is a valid move in $R_0^{||}$. According to the honest user strategy compilation, $skip(\kappa,\mathbb{B}) \in \Sigma_A^{||}(R^{||})$.

In both cases, $\delta \notin \Sigma_A^{||}(R_0^{||})$ so $\Sigma_A^{||} \not \models R^{||}$ which is a contradiction. We conclude then that $\pi.status \notin \{Choice, Left\}$.

We know by inductive hypothesis that if $(\forall B \in \mathcal{U}_{\kappa}(R^{||}).\pi.status \neq \mathbf{Assigned}(B)) \implies \mathbf{A} \notin \mathcal{R}^{ss}_{\kappa}(R^{||}).$

Now suppose that $ldesc(\kappa) \neq \emptyset$. We know by inductive hypothesis that $rdesc(\kappa) \neq \emptyset$ and also that $\mathcal{R}^{ss}_{\kappa_0}(R^{||}) \setminus \{A\} \neq \emptyset$. We can prove by contradiction that $\pi.status = \mathbf{CompensatedFrom}(B)$.

- We proved already that $\pi.status \notin \{Slashed(A), CompensatedFrom(A)\}$.
- If $\pi.status =$ Right, then $slash(\kappa, \mathbb{B}, \mathbb{B}) \in \Sigma_{\mathbb{A}}^{||}(R_0^{||}).$
- If $\pi.status = \mathbf{Slashed}(C)$ (where C is not necessarily B), then $slash(\kappa, \mathbb{B}, C) \in \Sigma_A^{||}(R_0^{||})$.

For the last two cases, this would imply again that $\delta \notin \Sigma_A^{||}(R_0^{||})$ so $\Sigma_A^{||} \notin R^{||}$ which is a contradiction.

• If $r_0 = |\kappa| \wedge p_0 = 1 \wedge r > |\kappa|$, then we know by inductive hypothesis that $\pi.status \notin \{ \text{Choice}, \text{Left}, \text{Slashed}(A), \text{CompensatedFrom}(A) \}.$ We can prove by contradiction that $\pi.status \neq \text{Right}$ and for every user B, $\pi.status \neq \text{Slashed}(B)$. Again, we will use the tactic of proving that there's another valid move that the honest user strategy would do instead of waiting which would imply that $\delta \notin \Sigma_A^{|I|}(R_0^{|I|})$ and $\Sigma_A^{|I|} \not\models R^{|I|}$

- If $\pi.status = \mathbf{Right}$, we can observe that by the timeout consistency invariant $\pi.time = t_0 + (2|\kappa| + 1)\hat{\delta}$, where t_0 is the stipulation time of the root contract of κ . We also know that κ is in skipping phase in $R_0^{||}$, so $r = |\kappa|$ and $(2r+1)\hat{\delta} \leq t t_0$ or, solving for $t, t \geq t_0 + (2|\kappa| + 1)\hat{\delta} = \pi.time$. Then, $right(\kappa, \mathbb{B})$ is a valid move in $R_0^{||}$ and $right(\kappa, \mathbb{B}) \in \Sigma_A^{||}(R_0^{||})$.
- If $\pi.status = \mathbf{Slashed}(B)$ for some B, then $compensate(\kappa, \mathbb{B}, B)$ is a valid move in $R_0^{||}$ and $compensate(\kappa, \mathbb{B}, B) \in \Sigma_A^{||}(R_0^{||})$.

Honest Skip

Let $\langle D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}$ with round status $(r_0, p_0) = \text{roundStatus}(\kappa, R_0^{||})$ before the δ move and $(r, p) = \text{roundStatus}(\kappa, R^{||})$ after it, such that after the wait move $r_0 < |\kappa| = r$.

We will prove by contradiction that $skip(\kappa) \in \Sigma_A^x(R_0^x)$. Suppose that the opposite is true and $skip(\kappa) \notin \Sigma_A^x(R_0^x)$. Then because Σ_A^x is eager, we know that either $\alpha(\kappa) \in \Sigma_A^x(R_0^x)$ for $\alpha \in \{dwithdraw, split, reveal\}$, $(A: \kappa) \in \Sigma_A^x$ or $A: s \in \Sigma_A^x(R_0^x)$. Then, according to the honest user strategy compilation:

- If $\alpha(\kappa) \in \Sigma_A^x(R_0^x)$, then $(A: s_{\kappa}^A) \in \Sigma_A^{||}(R_0^{||})$.
- If $(A : \kappa) \in \Sigma_A^x(R_0^x)$, then $(A : (\kappa, \mathbb{B})) \in \Sigma_A^{||}(R_0^{||})$.
- If $A: s \in \Sigma_A^x(R_0^x)$, then $A: s \in \Sigma_A^{||}(R_0^{||})$.

In all cases, the honest user strategy would not wait, so similarly to previous cases $\Sigma_A^{||} \not \models R^{||}$.

APPENDIX I INTERMEDIATE SEMANTICS TO BitML COHERENCE

A. Batch-Advertisments

In order to capture cross-chain contracts by compilation to BitML, we need to slightly generalize the BitML language and semantics. In particular, we will add the possibility to introduce contracts that share secrets in a batch, so as to give 'global' secrets that can be used by multiple contracts. The additions to the BitML syntax and semantics are minor.

We first define the notion of a batch advertisement.

Definition 14 (Batch advertisement). A batch advertisement is a term $\{S\}[\overrightarrow{\{G\}C}]$ where S is a possibly empty sequence of shared secret declarations A: secret S and S is a non-empty sequence of BitML contracts advertisements S is a possibly empty sequence of that the following conditions are satisfied:

- 1) Every contract advertisement $\{G_i|S\}C_i$ is a valid BitML advertisements
- 2) The deposit names among all preconditions G_i are distinct.
- 3) No additional local secrets in preconditions G_i
- 4) No volatile deposits in preconditions G_i
- 5) Every precondition G_i states the same set of participants.

Batch advertisements allow for the joint advertisement of contracts between the same set of users and shared secrets. We adopt the BitML stipulation rules in the following.

$$S = secrets(G^{\mathbb{B}})$$

$$a_{1} \dots a_{k} \text{ secrets of } A \text{ in } S$$

$$\forall i \in 1 \dots k. \quad \exists N. \quad \{A : a_{i} \# N_{i}\} \in \Gamma$$

$$\Delta = \underset{i=1,\dots,k}{\parallel} \quad \{A : a_{i} \# N_{i}\} \in \Gamma$$

$$\nabla i \in [1,k]. \quad N_{i} \in \begin{cases} \mathbb{N} & \text{if } A \in Hon \\ \mathbb{N} \cup \{\bot\} & \text{otherwise} \end{cases}$$

$$F \mid \underset{\mathbb{B} \in \mathcal{B}}{\parallel} \{G\}^{\mathbb{B}} C \xrightarrow{A:\{S\} \mid \overline{\{G\}^{\mathbb{C}}}, \Delta} \Gamma \mid \underset{\mathbb{B} \in \mathcal{B}}{\parallel} \{G\}^{\mathbb{B}} C \mid \Delta \mid \underset{\mathbb{B} \in \mathcal{B}}{\parallel} A \not = AuthCommit}$$

$$\forall A \in part(G). \quad A \not = \{G\}^{\mathbb{B}} C \mid C \mid \Gamma$$

$$(A : !v^{\mathbb{B}} @ x) \in G$$

$$\Gamma \mid \{G\}^{\mathbb{B}} C \xrightarrow{A:\{G\}^{\mathbb{B}} C), x} \Gamma \mid \{G\}^{\mathbb{B}} C \mid A \not = AuthInit$$

$$G = (\underset{i \in I}{\parallel} A_{i} : !v_{i}^{\mathbb{B}} @ x_{i}) \mid (\underset{j}{\parallel} C_{j} : secret \quad a_{j})$$

$$x \text{ fresh}$$

$$\Gamma \mid \{G\}^{\mathbb{B}} C \mid (\underset{i \in I}{\parallel} \langle A_{i}, v_{i} \rangle_{x_{i}}^{\mathbb{B}}) \mid (\underset{A \in G}{\parallel} A \not = A \not= A \not = A \not= A \not = A \not= A$$

The introduction of batch advertisements induces a small modification in the stipulation protocol. The following description of the new stipulation protocol heavily relies on notion and definition introduced in the BitML paper [1].

Definition 15 (Stipulation Protocol). Let $A \in Hon$, let R_*^C be a (A-stripped) computational run, and let r_A be a random sequence. The stipulation protocol for a computational batch contract advertisement C_{batch} is as follows.

- 1) A decodes C_{batch} , constructing a symbolic batch contract advertisement $\{S\}[\{G\}\acute{C}]$; in doing this; A chooses distinct symbolic names for all the transaction outputs in C_{batch} . The mapping txout is defined according to the used correspondence between names and transaction outputs.
- 2) A infers from \vec{G} the parameters \vec{part} , $\vec{Part}\vec{G}$ and \vec{val} where $\vec{Part}\vec{G}$ refers to the participants in any $G \in \vec{G}$ (not that all preconditions are required to feature the same set of participants).
- 3) A uses r_A to obtain the key pairs $K_A(r_A)$ and $\hat{K}_A(r_A)$. The key $K_A(r_A)$ is used by A to sign all the protocol messages. Further, A reads from the initial prefix of the run R_*^C , the public keys $K_B^p(r_B)$ and $\hat{K}_B^p(r_B)$ of all the $B \in Part\vec{G}\setminus\{A\}$. The key $K_B^p(r_B)$ is used by A to filter out the incoming messages with incorrect signatures.
- 4) A generates from r_A a secret nonce of desired length for each $A: a_i \in S$. Then A computes the hashes $\vec{h} = h_1, \ldots, h_k$ of secret nonces (by querying O), and broadcasts $m^*(\mathcal{C}_{batch}, \vec{h})$. Dually, A receives the hashes \vec{h}' from the other participants. When doing so, A starts defining sechash using the first (correctly signed) $m^*(\mathcal{C}_{batch}, \vec{h}')$ in R^C_* which has no duplicate hashes, and has no hashes already occurring (signed) in R^C_*
- 5) After receiving the corresponding hashes of all other participants, for each $G_i \in \vec{G}$, A performs steps (4) and (6) to (8) of the original stipulation protocol. Instead of the message $m(C, \vec{h}, \vec{k})$, messages of the form $m(C_{batch}, C, \vec{h}, \vec{k})$ are used to identify the contract advertisement C within the batch advertisement C_{batch} . Step (5) is skipped as local secrets are not supported (and needed) in our model.

Note that as opposed to the original stipulation protocol in [1], we slightly adapted the order of message exchanges: First, the common secrets are committed and exchanged. At this point, the compiler keys are not generated and exchanged yet. This, however, should not cause an issue since the compiler keys can also be generated in a later step and broadcasted jointly with the contract-specific secrets.

The adaptation of the proof follows naturally since we simply add a first authorization phase where the users commit to their secrets. Intuitively, the broadcast of messages of the form $m^*(\mathcal{C}_{batch}, \vec{h})$ now reflect the *Batch-AuthCommmit* steps, while broadcasts of messages of the form $m(\mathcal{C}_{batch}, \mathcal{C}, \vec{h}, \vec{k})$.

B. Low Level Coherence

Instead of defining a BitML to intermediate level coherence as before, we are defining a function mapping intermediate semantics actions and configurations to corresponding BitML actions and configurations.

- 1) Intermediate Level Configurations: Individual components of an intermediate configuration $\Gamma^{||}$ can be translated to BitML configuration components with the following function $f_{c,\Gamma^{||}}()$. \mathcal{F}_{Stip} , \mathcal{F}_{Compen} , \mathcal{F}_{Refund} and \mathcal{F}_{C} are invocation of the $BitML^{x}$ compiler.
 - $f_{c,\Gamma||}(\{G^x\}^{\mathbb{B}}C^x) = \mathcal{F}_{Adv}(\{G^x\}C^x)\downarrow^{\mathbb{B}}$ where $\downarrow^{\mathbb{B}}$ return the precondition-contract component of blockchain \mathbb{B}
 - $f_{c,\Gamma||}(A[\# \triangleright \{S\}[\overline{\{G\}C}]]) = A[\# \triangleright \mathcal{F}_{Adv}(\{G^x\}C^x) \downarrow^{\mathbb{B}}]$
 - $f_{c,\Gamma^{\parallel}}(A[x \triangleright \{G^x\}^{\mathbb{B}}C^x]) = A[x \triangleright \mathcal{F}_{Adv}(\{G^x\}C^x) \downarrow^{\mathbb{B}}]$
 - Let Ω be the compiler settings of the contract κ restored from $R^{||}$. Let G^x be the preconditions of the contract κ restored from $R^{||}$. Let the default balance v be $\pi[balance]$ ||||and <math>n be the number of participants in a contract κ .

om
$$R^{||}$$
. Let the default balance v be $\pi[balance[B]]$ and n be the number of participants in a contract κ .
$$\begin{cases} \langle \mathcal{F}_{Stip}(\{G^x\}C^x), v\rangle_x^{\mathbb{B}} & \text{if } \pi[status] = \mathbf{Stip\text{-}Choice} \\ \langle \mathcal{F}_{Stip}(\{G^x\}C^x) \xrightarrow{\tau}, v\rangle_x^{\mathbb{B}} & \text{if } \pi[status] = \mathbf{Stip\text{-}Right} \\ \langle \mathcal{F}_{Compen}^{\mathbb{B}}(\Omega) \xrightarrow{put(\emptyset, \{s_{\lambda}^A\}, x_0)^{\mathbb{B}}}, v\rangle_x^{\mathbb{B}} & \text{if } \pi[status] = \mathbf{Stip\text{-}Slashed}(A) \\ \langle \mathcal{F}_{Compen}^{\mathbb{B}}(\Omega) \xrightarrow{put(\emptyset, \{s_{\lambda}^A\}, x_{-1})^{\mathbb{B}}} \xrightarrow{split(x_0)^{\mathbb{B}}}, v\rangle_{x_i}^{\mathbb{B}} & \text{if } \pi[status] = \mathbf{Stip\text{-}Compensation}(A) \\ \langle \mathcal{F}_{C}^{\mathbb{B}}(\mathcal{F}_{Refund}(G^x), \Omega), v\rangle_x^{\mathbb{B}} & \text{if } \pi[status] = \mathbf{Choice} \\ \langle \mathcal{F}_{C}^{\mathbb{B}}(C^x, \Omega), v\rangle_x^{\mathbb{B}} & \text{if } \pi[status] = \mathbf{Left} \\ \langle \mathcal{F}_{C}^{\mathbb{B}}(C^x, \Omega) \xrightarrow{put(\emptyset, \{s_{\lambda}^A\}, x_0)^{\mathbb{B}}}, v\rangle_x^{\mathbb{B}} & \text{if } \pi[status] = \mathbf{Right} \\ \langle \mathcal{F}_{Compen}^{\mathbb{B}}(\Omega) \xrightarrow{put(\emptyset, \{s_{\lambda}^A\}, x_0)^{\mathbb{B}}}, v\rangle_x^{\mathbb{B}} & \text{if } \pi[status] = \mathbf{Choice} \\ \langle \mathcal{F}_{Compen}^{\mathbb{B}}(\Omega) \xrightarrow{put(\emptyset, \{s_{\lambda}^A\}, x_0)^{\mathbb{B}}}, v\rangle_x^{\mathbb{B}} & \text{if } \pi[status] = \mathbf{Slashed}(A) \\ \langle \mathcal{F}_{Compen}^{\mathbb{B}}(\Omega) \xrightarrow{put(\emptyset, \{s_{\lambda}^A\}, x_0)^{\mathbb{B}}}, v\rangle_x^{\mathbb{B}} & \text{if } \pi[status] = \mathbf{CompensatedFrom}(A) \\ \langle withdrawA, v\rangle_x^{\mathbb{B}} & \text{if } \pi[status] = \mathbf{Assigned}(A) \\ \text{here the notion } \langle C \xrightarrow{\alpha}, v\rangle_x^{\mathbb{B}} & \text{is defined as} \\ \end{pmatrix}$$

where the notion $\langle C \xrightarrow{\alpha}, v \rangle_{x}^{\mathbb{B}}$ is defined as

$$\langle C \xrightarrow{\alpha}, v \rangle_x^{\mathbb{B}} := \langle C', v \rangle_x^{\mathbb{B}} \text{ if } (\langle C, v \rangle_{x_0}^{\mathbb{B}} \xrightarrow{\alpha} \langle C', v \rangle_x^{\mathbb{B}}$$

with the special case of splits, where

$$\langle split \ \vec{v} \rightarrow \vec{C} \xrightarrow{split(x_0)^{\mathbb{B}}}, \vec{v} \rangle_{\vec{x}}^{\mathbb{B}} \ := \ \prod_{i=1}^{\rightarrow} \langle C_i, v_i \rangle_{x_i}^{\mathbb{B}}$$

- $f_{c,\Gamma||}(A_i[(\kappa,\mathbb{B},x) \triangleright D^x]) = A_i[x \triangleright \mathcal{F}_D^{\mathbb{B}}(D, \Omega(\Gamma||,\kappa))]^{\mathbb{B}}$
- $f_{c,\Gamma||}(\underline{\ })$ maps everything related to secrets and deposits to itself: $f_{c,\Gamma^{||}}(id) = id$ if

$$id \cong A : s\#N \lor id \cong \{A : s\#N\} \lor id \cong \{A : s_{\kappa}^{A}\} \lor id \cong A : s_{\kappa}^{A} \lor id \cong \{A : is_{\kappa}^{A}\} \lor id \cong A : is_{\kappa}^{A} \lor id \cong \langle A, v \rangle_{\mathbb{R}}^{B} \lor id \cong A : x, j$$

- 2) Intermediate to BitML Action Translation: CA on possible moves of $\Sigma_A^{||}(R^{||})$:
- Advertise: $f_a(advertise(\{G^x\}C^x)) = \mathcal{F}_{Adv}(\{G^x\}C^x)$
- Authorize Commit: $f_a(commit(A, \{G^x\}C^x)) = A : \mathcal{F}_{Adv}(\{G^x\}C^x), \Delta$
- Authorize Init: $f_a(authInit(A, \{G^x\}^{\mathbb{B}}C^x)) = A : \{G\}^{\mathbb{B}}C, x(A, G) \text{ where } \{G\}^{\mathbb{B}}C = \mathcal{F}_{Adv}(\{G^x\}C^x) \downarrow^{\mathbb{B}}C$
- Publish: $f_a(publish(\{G^x\}^{\mathbb{B}}C^x)) = init(G,C)^{\mathbb{B}}$ where $\{G\}^{\mathbb{B}}C = \mathcal{F}_{Adv}(\{G^x\}C^x) \downarrow^{\mathbb{B}}$
- Reveal Init Step Secret: $f_a(A:is_{\kappa}^A) == A:s_{\kappa}^A$
- Init: $f_a(init(\kappa, \mathbb{B}, x)) = put(\emptyset, \{s_{\lambda}^{\mathbf{A}}\} \cup S, x)^{\mathbb{B}}$
- Stip Skip: $f_a(from\ t:\ sskip(\kappa,\mathbb{B})) = A: is_{\kappa}^A$
- Abort: $f_a(from\ t:\ abort(\kappa,\mathbb{B},x)) = put(\emptyset,\emptyset,x)^{\mathbb{B}}$
- Intro Stip Compensate: $f_a(sslash(\kappa_0, \mathbb{B}, A, x)) = put(\emptyset, \{s_{\lambda}^{A}\}, x)^{\mathbb{B}}$
- Elim Stip Compensate: $f_a(scompensate(\kappa_0, \mathbb{B}, A, x)) = split(x)$
- Secret Reveal: $f_a(\mathbf{A}:a) = \mathbf{A}: A\kappa$
- Step Secret Reveal: $f_a(A: s_{\kappa}^A) = A: s_{\kappa}^A$
- Intro Left: $f_a(ileft(\kappa, \mathbb{B}, x)) = put(\emptyset, \{s_{\lambda}^{\mathbb{A}}\}, x)^{\mathbb{B}}$
- Reveal: $f_a(reveal(\kappa, \mathbb{B}, x)) = put(\emptyset, \{s_{\lambda}^{A}\} \cup \vec{s}, x)^{\mathbb{B}}$
- Authorize: $f_a(\mathbf{A}:(\kappa,\mathbb{B},x)) = \mathbf{A}:x,D(\mathbb{R}^{||},\kappa)^{\mathbb{B}}$
- Intro Skip: $f_a(from\ t:\ skip(\kappa,\mathbb{B},x))=put(\emptyset,\emptyset,x)^{\mathbb{B}}$

- Elim Skip: $f_a(from\ t:\ right(\kappa,\mathbb{B},x))=put(\emptyset,\emptyset,x)^{\mathbb{B}}$
- Intro Compensation: $f_a(slash(\kappa, \mathbb{B}, x, A)) = put(\emptyset, \{s_{\lambda}^{A}\}, x)^{\mathbb{B}}$
- Elim Compensation: $f_a(compensate(\kappa, \mathbb{B}, x, A)) = split(x_0)$
- Guarded Withdraw: $f_a(dwithdraw(\kappa, \mathbb{B}, x)) = split(x)^{\mathbb{B}}$
- Split: $f_a(split(\kappa, \mathbb{B}, x)) = split(x)^{\mathbb{B}}$
- Withdraw: $f_a(cwithdraw(\kappa, \mathbb{B}, x)) = split(x)^{\mathbb{B}}$

where

- x(A,G) returns the identifier x for participants A deposit in preconditions G
- $D(R^{||}, \kappa)^{\mathbb{B}}$ returns a contract branch of contract κ in $\Gamma^{||}(R^{||})$ with an authorization (note: there will only be a single branch with authorization in practice)

Non contract-related cases:

- Time: $f_a(\delta) = \delta$
- Double Spend: $f_a(doublespend(\{G^x\}^{\mathbb{B}}C^x, A, x)) = destroy([x])$
- Auth Double Spend: $f_a(A:doublespend(\{G^x\}^{\mathbb{B}}C^x,A,x))=A:x,j$

Definition 16 (BitML Strategy Compilation). We define the compiled low-level honest participant strategy for two coherent runs $R^{||} \approx R$ with strategy $\Sigma_A = \mathcal{S}^{||}(\Sigma_A^{||})$ for participant A as

$$f_a(\alpha) \in \Sigma_{\mathbf{A}}(R) \iff \alpha \in \Sigma_{\mathbf{A}}^{||}(R^{||})$$

3) Translation Correctness:

Lemma 13 (Translation Correctness I).

$$\forall \Gamma_0^{||} \beta. \ \Gamma_0^{||} \xrightarrow{\beta} \Gamma^{||} \Rightarrow f_c(\Gamma_0^{||}) \xrightarrow{f_a(\beta)} f_c(\Gamma^{||})$$

Proof. Formal proof by case analysis over all possible intermediate semantics actions β . The semantics rule of each action β implies the existence of certain intermediate semantics configuration elements $\gamma \in \Gamma_0^{||}$ that enabled β . We prove for each case, that the set of BitML configuration elements consisting out of translated intermediate semantics configuration elements (i.e., $f_{c,\Gamma^{||}}(\gamma)$), is enough to show the existence of $f_a(\beta)$. Additionally, it is shown in each case that the consumed and added configuration elements coincide.

Lemma 14 (Translation Correctness II).

$$\forall \Gamma_0^{||} \alpha. \ f_c(\Gamma_0^{||}) \xrightarrow{\alpha} \ \land \ \alpha \not\cong with draw \ \Rightarrow \ \exists \beta. f_a(\beta) = \alpha \ \land \ \Gamma_0^{||} \xrightarrow{\beta}$$

Proof. Formal proof by case analysis over all possible BitML semantics actions α . Every BitML action α implies certain elements γ in the configuration, for which another case distinction over $f_{c_{\Gamma_{0}^{||}}}(\gamma)$ is made. This second case distinction restricts α to contracts from one of the $BitML^{x}$ compilers (as defined in $f_{c_{R||}}$). Lastly, we check that for each case left, there exists an action $\beta \in f_{c,\Gamma||}(\alpha)^{-1}$. There is a restriction on withdraws as they can not (and don't need to) be scheduled in the intermediate semantics.

Lemma 15 (BitML Configuration Consistency).

$$\forall \Gamma_0, \Gamma'_0. \ \Gamma_0 \xrightarrow{\alpha} \Gamma \ \land \ \Gamma_0 \subseteq \Gamma'_0 \ \Rightarrow \ \Gamma'_0 \xrightarrow{\alpha} \Gamma' \ \land \ \Gamma \subseteq \Gamma'$$

Proof. Trivial by case analysis on BitML semantics.

Lemma 16 (Synchronous Actions). Execution of BitML actions in two runs low-level R_0 and R'_0 changes configuration elements synchronously (i.e., adding/deleting the same elements to/from the configuration) as long as contract identifiers and deposit- and secret names have the same meaning.

Let R_0 and R'_0 be two BitML runs where contract identifiers and deposit- and secret names have the same meaning then

$$R_0 \xrightarrow{\alpha} R \wedge R'_0 \xrightarrow{\alpha} R' \Rightarrow \exists \Gamma^-, \Gamma^+. \Gamma(R) = \Gamma_0 \cup \Gamma^+ \setminus \Gamma^-$$

 $\wedge \Gamma(R') = R_0 \cup \Gamma^+ \setminus \Gamma^-$

Proof. Induction over bitml semantics action α .

4) Low Level Coherence: Initial configuration consists only out of deposits. To simplify the presentation of the proof, we are abstracting parts of BitML deposit handling infrastructure, i.e. we only allow the destruction of deposit and disregard joining, dividing and donating deposits. In the intermediate semantics the destruction of deposits is dubbed under the umbrella term doubleSpend.

$$\begin{split} &\Gamma^{||}(R_0^{||}) = \Gamma_0^{||} \ initial \\ &\frac{\Gamma(R_0) = \Gamma_0 = f_c(\Gamma_0^{||})}{R_0^{||} \approx R_0} \approx -Initial \\ &\frac{R_0^{||} \approx R_0}{R_0} \approx R \\ &R = R_0 \xrightarrow{\alpha} \Gamma \\ &R^{||} = R_0^{||} \xrightarrow{\beta} \Gamma^{||} \\ &\frac{f_a(\beta) = \alpha}{R^{||} \approx R} \approx -Valid \\ &R_0^{||} \approx R_0 \\ &R = R_0 \xrightarrow{\alpha} \Gamma \\ &\frac{\not\exists \beta. f_a(\beta) = \alpha}{R_0^{||} \approx R} \approx -Invalid \end{split}$$

5) Soundness:

Lemma 17 (Soundness of Intermediate to Low-level Compilation). Let Σ_A be a compiled $BitML^x$ intermediate semantics strategy $\Sigma_A = \mathcal{S}^{||}(\Sigma_A^{||})$. The five conditions after strategy conformance and coherence are called coherence invariants.

$$\begin{split} \forall R. \ \Sigma_{A} \vdash R \Rightarrow \exists R^{||}. \ R^{||} \approx R \ \land \ \Sigma_{A}^{||} \vdash R^{||} \\ \land \ \forall A \in Hon. \ A: \{G\}C, \Delta_{0} \in R \Rightarrow \forall A: \{G\}C, \Delta \in R \\ (\forall \{s\} \in \Delta: \{s\} \in R \Rightarrow \{s\} \in f_{c}(R^{||})) \\ s \in \Gamma(R) \Rightarrow s \in f_{c}(\Gamma^{||}(R^{||}))) \\ \forall A: \{G\}C, \Delta \in \Gamma(R) \Rightarrow A: \{G\}C, \Delta \in f_{c}(\Gamma^{||}(R^{||})) \\ \forall A: \{G\}^{\mathbb{B}}C, x \in \Gamma(R) \Rightarrow A: \{G\}^{\mathbb{B}}C, x \in f_{c}(\Gamma^{||}(R^{||})) \\ \land \ \forall \langle C, v \rangle_{x}^{\mathbb{B}} \in \Gamma(R) \Rightarrow \langle C, v \rangle_{x}^{\mathbb{B}} \in f_{c}(\Gamma^{||}(R^{||})) \\ \land \ \forall A: x, D \in \Gamma(R) \Rightarrow A: x, D \in f_{c}(\Gamma^{||}(R^{||})) \\ \land \ \forall \langle A, v \rangle_{x}^{\mathbb{B}} \in initial(\Gamma(R)). \ \langle A, v \rangle_{x}^{\mathbb{B}} \in \Gamma(R) \Rightarrow \langle A, v \rangle_{x}^{\mathbb{B}} \in f_{c}(\Gamma^{||}(R^{||})) \\ \land \ \forall \langle A, v \rangle_{x}^{\mathbb{B}} \in \Gamma. \ \neg initial(\langle A, v \rangle_{x}^{\mathbb{B}}) \Rightarrow \langle withdrawA, v \rangle_{x}^{\mathbb{B}} \in f_{c}(\Gamma^{||}(R^{||})) \end{split}$$

Proof. Induction over trace R with trivial base case.

Induction step $R_0 \xrightarrow{\alpha} \Gamma$. Case analysis on \approx rule condition:

- $\exists \beta. f_a(\beta) = \alpha \land R_0^{||} \xrightarrow{\beta} \Gamma^{||}$: Application of $\approx -Valid$ rule. Strategy compliance follows directly by definition of strategy compilation. All invariants hold with the synchronous actions lemma (Lemma 16) and the inductive hypothesis. The only special case is the [C-AuthCommit] case where it needs to be reasoned that commitments made prior to the first honest user commitment are present in $f_c(R^{||})$. This is the case, since all valid stipulation commitments are enabled with the only configuration precondition being that the advertisement is present. Advertisements are present in $f_c(R^{||})$ as long the advertisement is a valid $BitML^x$ contract which is implied by an honest user trying to authorize it (by the definition of strategy compilation).
- $\not\exists \beta. f_a(\beta) = \alpha \land R_0^{||} \xrightarrow{\beta} \Gamma^{||}$: Application of $\approx -Invalid$ rule. Strategy compliance is correct by induction hypothesis. Proof by case analysis on BitML semantics rule emitting α .
 - Contract Rules: [C-Split], [C-PutRev], [C-AuthControl] In each rule, the inductive hypothesis implies that all precondition that were present in $\Gamma(R_0)$ must also be present

in $f_c(\Gamma^{||}(R_0^{||}))$ (since every contract has an honest user, all conditions in the hypothesis is met). Hence, the step $f_c(\Gamma^{||}(R_0^{||})) \stackrel{\alpha}{\to}$ is enabled as well. The second Translation Correctness lemma (Lemma 14) of f_c guarantees that in that case, there is also at least one enabled action in $R_0^{||}$. This contradicts the case analysis assumption.

- Secret [C-AuthRev]
 - Either the revealed secret was not part of $f_c(R^{||})$ in the first place or it was part of $f_c(R^{||})$ in which case the action could have been scheduled as well.
- Stipulation: [C-Advertise]
 - Advertisments can only not be scheduled in the intermediate semantics when they are not a valid $BitML^x$ contracts or missing non-initial deposits. The relevant parts of the active configuration do not change. Hence, the invariants holds by inductive hypothesis.
- Stipulation: [C-AuthCommit]
 - Stipulation commitment authorizations can only not be scheduled in $\Gamma^{||}(R_0^{||})$ if the contract advertisement is not present. Without contract advertisement in the intermediate run, no honest user can commit to this contract. Hence, the invariants holds by inductive hypothesis.
- Stipulation: [C-AuthInit]
 - Stipulation init authorizations can only not be scheduled in $\Gamma^{||}(R_0^{||})$ when the contracts advertisement or authorization commitments are missing. This case is proven by case distinction on whether the authorized contract was a $BitML^x$ contract or not. If yes, we show that this move was indeed possible as an honest user has committed to it and the move would have been enabled in $f_c(R^{||})$ as well. If not, no honest user committed, hence, the precondition that at least one honest participant committed is not met in R.
- Stipulation: [C-Init]
- Same reasoning as in the [C-AuthInit] case.
- Withdrawel: [C-Withdraw]
 - Withdraws cannot be scheduled in the intermediate semantics. Since the rule adds a non-initial deposit, it needs to be proven that a withdraw contract is part of the intermediate configuration. This follows from the inductive hypthesis and analysis of $f_c^{-1}(\langle withdraw_-, v \rangle_x^{\mathbb{B}})$.

APPENDIX J MONEY PRESERVATION

Definition 17 (Intermediate User Payouts). Let $R^{||}$ be an intermediate semantics run. We define the payout of participant A in blockchain \mathbb{B} (written $payout_A^{\mathbb{B}}(R^{||})$) as follows

$$\sum_{\{\pi: \langle C, \pi \rangle_{\kappa}^{\underline{B}} \in \Gamma_{R^{||}} \land \pi[status] = \textbf{Assigned}(\underline{A})\}} \pi.balance$$

Definition 18 (BitML^x User Payouts). Let R^x be a BitML^x run. We define the payout of participant A in R^x and blockchain B (written payout A (R^x)) as follows

$$\sum_{\langle A, \mathsf{B}
angle_\kappa \in \Gamma_{R^x}} \mathsf{B}[\mathbb{B}]$$

Definition 19 (Intermediate Active Contract Funds). Let $R^{||}$ be an intermediate semantics run. We define the contract funds in blockchain \mathbb{B} (written contractFunds $R^{||}(R^{||})$) as follows

$$\sum_{\substack{\pi. balance \\ \{\pi: \langle C, \pi \rangle_{\kappa}^{\mathbb{S}} \in \Gamma_{R|l} \land \pi[status] \neq \textbf{Assigned(A)}\} \land \pi[status] \neq \textbf{CompensatedFrom(A)}\}}}$$

Definition 20 (Intermediate Total Contract Funds). Let F be the maximal frontier for blockchain \mathbb{B} in the intermediate semantics run $R^{||}$ (maxFront $(F, R^{||}, \mathbb{B})$). We define the contract funds in F (written totalFunds $R^{||}$) as follows

$$\sum_{\{\pi: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R||}\}} \pi.balance$$

Definition 21 (BitML^x Active Contract Funds). Let F be the maximal frontier for BitML^x run R^x . We define the contract funds in Γ^x and blockchain $\mathbb B$ (written contractFunds R^x) as follows

$$\sum_{\left\{ \mathbf{B}:\left\langle C,\mathbf{B}\right\rangle _{\kappa}\in\Gamma_{R^{x}}\right\} }\mathbf{B}\left[\mathbb{B}\right]$$

Definition 22 (Intermediate User Inputs). Let $R^{||}$ be an intermediate semantics run. We define the intermediate inputs of A in $R^{||}$ and blockchain B (written inputs $A(R^{||})$) as follows

$$\sum_{\{v:A:!\,v\mathbb{B}\;@\,x\in G^{\mathbb{B}}\;\wedge publish(\{G^{\mathbb{B}}\}C)\in R^{||}\}}v$$

Definition 23 (BitML^x User Inputs). Let R^x be a BitML^x run. We define the inputs of A in R^x and blockchain \mathbb{B} (written inputs $A = R^x$) as follows

$$\sum_{\left\{ \text{B}: A: \text{!B} \ @ \ \vec{x} \in G \land init(\left\{G\right\}C) \in R^x \right\}} \text{B}\left[\mathbb{B}\right]$$

Definition 24 (Intermediate Compensated Funds). We define the compensated funds in \mathbb{B} of (written compFunds $(R^{||})$) as follows

$$\sum_{\{\pi: \langle C, \pi \rangle_{\kappa}^{\underline{\mathbf{n}}} \in \Gamma_{R} || \; \land \pi[status] = \textit{CompensatedFrom}(A)\}} \pi.balance$$

Lemma 18 (Intermediate Money Preservation). Let $R^{||}$ be an intermediate semantics run. Then, for every blockchain \mathbb{B} , the following equation holds

$$\mathit{totalFunds}^{\mathbb{B}}(R^{||}) = \sum_{A \in P} \mathit{inputs}^{\mathbb{B}}_{A}(R^{||})$$

Proof. By induction over $R^{||}$. For the base case of the empty configuration, both sides are 0.

Let's first consider the case of a move affecting an active contract. Let $R^{||} = R_0^{||} \xrightarrow{\alpha(\kappa,\mathbb{B})} \Gamma^{||}$ with the move $\alpha(\kappa,\mathbb{B})$ resulting in successor contracts $\vec{\kappa}$, and let F_0,F be frontiers such that $\max Front(F_0,R_0^{||},\mathbb{B})$ and $\max Front(F,R^{||},\mathbb{B})$.

We know by inductive hypothesis that

$$\mathsf{totalFunds}^{\mathbb{B}}(R_0^{||}) = \sum_{A \in P} \mathsf{inputs}_A^{\mathbb{B}}(R_0^{||})$$

Given that we are doing an $\alpha(\kappa, \mathbb{B})$ move, we know that $\kappa \in F_0$. Then, by the blockchain frontier update lemma $F = F_0 \setminus \{\kappa\} \cup \{\kappa^{\downarrow} \in \vec{\kappa}\}$. This move also is also not a *publish* move so user inputs will remain constant. If we assume that the balance of κ is v_{κ} and the successors $\kappa_1, \ldots, \kappa_n \in \vec{\kappa}$ have balances v_1, \ldots, v_n respectively, then we have:

$$\begin{split} &\mathsf{totalFunds}^{\mathbb{B}}(R^{||}) \\ &= \mathsf{totalFunds}^{\mathbb{B}}(R_0^{||}) - v + \sum_{i=1}^n v_i \\ &= \mathsf{inputs}^{\mathbb{B}}_A(R_0^{||}) - v + \sum_{i=1}^n v_i \\ &= \mathsf{inputs}^{\mathbb{B}}_A(R^{||}) - v + \sum_{i=1}^n v_i \end{split}$$

For our goal condition to hold, we only need to prove that $v = \sum_{i=1}^{n} v_i$, which follow immediately from the transition rules for each case of the intermediate semantics moves.

Lemma 19 (BitML^x Money Preservation). Let F be the maximal frontier for BitML^x run R^x . Then, the following equation holds:

$$\sum_{A \in P} \textit{payout}_A^{\mathbb{B}}(R^x) + \textit{contractFunds}^{\mathbb{B}}(R^x) = \sum_{A \in P} \textit{inputs}_A^{\mathbb{B}}(R^x)$$

APPENDIX K LIQUIDITY

A. Eager and deterministic strategies

We will assume that $BitML^x$ strategies are eager meaning that a user will always schedule a contract action if it is possible. This implies in particular that a user will schedule a skip action whenever the priority action is "blocked" by another user.

Definition 25 (Eager BitML^x strategies). A BitML^x strategy Σ_A^x is eager if the following condition holds:

$$\forall R^x. \ \Sigma_{\mathbf{A}}^x \vdash R^x \Rightarrow \forall \kappa \in contracts(R^x): (\exists \alpha \Gamma^x: R^x \xrightarrow{\alpha(\kappa)} \Gamma^x) \Rightarrow \exists \alpha' \in \Sigma_{\mathbf{A}}^x(R^x) \\ \wedge \ (\exists \hat{\alpha}(\kappa). \ \hat{\alpha}(\kappa) = \alpha' \ \lor \ \alpha' = \mathbf{A}: \kappa \\ \vee \ (\langle \mathit{reveal} \ S \ \mathit{then} \ C \ \Leftrightarrow D, \mathbf{B} \rangle_\kappa) \in \Gamma^x \ \wedge \ \alpha' = \mathbf{A}: a \ \wedge \ a \in S) \\ \wedge \ (\forall \Gamma^x: R^x \xrightarrow{\alpha'} \Gamma^x \Rightarrow \Gamma^x \neq \Gamma^x(R^x))$$

Note that this definition ensures that the honest user strategy needs to eventually schedule skip actions (if possible) but at the same time allows for scheduling other actions α' , e.g., authorizations or reveal secrets for enabling other (left) moves. However, it is required that such α' actions need to change configurations (this, in particular, forbids double authorizations).

In addition to strategies being eager, we will also require them to be *deterministic*. Intuitively, a strategy is deterministic if will never schedule contradicting actions for the same contract.

Definition 26 (Deterministic BitML^x strategies). A BitML^x strategy Σ_A^x is deterministic if the following condition holds:

$$\forall R^x. \ \Sigma_A^x \vdash R^x \Rightarrow \forall \alpha(\kappa), \alpha'(\kappa) \in \Sigma_A^x(R^x) \Rightarrow \alpha = \alpha'$$

Note that Σ_A^x being deterministic and persistent ensures in particular that it will never happen that Σ_A^x schedules a $skip(\kappa)$ action and then a $left(\kappa)$ action afterwards.

Determinism is from a security perspective a sensible requirement on user strategies, since in cases of non-determinism, it is left to the (adversarial) scheduler to resolve non-determinism.

Another requirement on x-strategies $\Sigma_{\mathbf{A}}^{x}$ will be that those strategies will only propose and execute a finite number of contracts \mathcal{K} . This requirement is needed since, technically it is not prevented that an Σ_A^x keeps on proposing (the same) contracts over and over again. We will call such strategies to be K-bounded.

Definition 27 (x-K-Boundedness). Let A be an honest participant with strategy Σ_A^x and K be a set of contracts. We call Σ_A^x bounded by K if for all runs R^x with $\Sigma_A^x \vdash R^x$ it holds that if $advertise(G,C) \in \Sigma_A^x(R^x)$ and $(t_0,\kappa_0) = initSettings(G)$

- 1) $\kappa_0 \in \mathcal{K}$
- 2) $\kappa_0 \notin contracts_{adv}(R^x)$

We define a similar notion for the intermediate level:

Definition 28 (Intermediate K-Boundedness). Let A be an honest participant with strategy $\Sigma_A^{||}$ and K be a set of contracts. We call $\Sigma_A^{||}$ bounded by K if for all runs $R^{||}$ with $\Sigma_A^{||} \vdash R^{||}$ it holds that if $advertise(\{G\}C) \in \Sigma_A^{||}(R^{||})$ and $(t_0, \kappa_0) = 1$ initSettings(G) then

- 1) $\kappa_0 \in \mathcal{K}$
- 2) $\kappa_0 \notin contracts_{adv}(R^{||})$

Lemma 20. Let A be an honest participant with strategy Σ_A^x and $\Sigma_A^{||} = \mathcal{S}(\Sigma_A^x)$ its corresponding intermediate strategy. Further let K be a set of contracts such that Σ_A^x is K-bounded. Then also $\Sigma_A^{||}$ is K-bounded.

We will define Liquidity similar to how it is defined in [11] and show that all $BitML^x$ contracts are liquid w.r.t. eager

Next, we will show how to prove liquidity for compiled (eager) strategies.

B. $BitML^x$ Liquidity

We adapt the definition from [11] to our setting:

Definition 29 (Strong BitML^x Liquidity). Let A be an honest participant with strategy Σ_A^x . Then Σ_A^x is strongly liquid if for all runs R^x with $\Sigma_A^x \vdash R^x$ there exists an extension $\hat{R}^x = R^x \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} \text{ of } R^x$ such that

- 1) $\forall i \in 1 \dots n : \alpha_i \in \Sigma_A^x(R^x \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{i-1}})$
- 2) $\not\exists \langle C, \mathsf{B} \rangle_{\kappa} \in \Gamma^x(\dot{R}^x)$

In contrast to the BitML-style liquidity notion, this notion takes into account that the strategy Σ_A^x will only advertise and execute contracts that they can also liquidate.

Lemma 21 (Non-blocking $BitML^x$ contracts). Given an $BitML^x$ run R^x , we can proof that every active $BitML^x$ contract has at least a single active move:

$$\forall \langle C, \mathbf{B} \rangle_{\kappa} \in \Gamma^x(R^x).(\exists \alpha \Gamma^x. R^x \xrightarrow{\alpha(\kappa)} \Gamma^x)$$

Proof. Proven by CA on *BitML*^x contract semantics using rules [C-Withdraw] and [C-Skip]

Definition 30 (BitML^x Liquidation Distance). Let A be an honest participant with strategy Σ_A^x and K a set of contracts such that Σ_A^x is K-bounded. The liquidation distance of an BitML^x run R^x w.r.t. to Σ_A^x (written $\|(R^x, \Sigma_A^x)\|$) is formally defined as follows:

$$\|(R^x, \Sigma_A^x)\| := (\mathcal{K}^-, \mathcal{C}_{adv}^-, \mathcal{A}_{adv}^-, \delta_{adv}, D, \mathcal{A}_{\mathcal{B}}^-, \mathcal{S}_{reveal}^-)$$

where

$$\mathcal{K}^- := \mathcal{K}/contracts_{adv}(R^x)$$

$$\mathcal{C}^-_{adv} := \{\kappa_0 \mid \{G\}C \in \Gamma^x(R^x) \ \land \ (t_0,\kappa_0) = initSettings(G) \ \land \ A : [\# \rhd \{G\}C \not\in \Gamma^x(R^x)\} \}$$

$$\mathcal{A}^-_{adv} := \{\kappa_0 \mid \{G\}C \in \Gamma^x(R^x) \ \land \ (t_0,\kappa_0) = initSettings(G) \ \land \ A : !\mathbb{B} @ \vec{x} \in G$$

$$\land \ A : [\vec{x} \rhd \{G\}C] \not\in \Gamma^x(R^x) \}$$

$$\delta_{adv} := \{\kappa_0 \mid \{G\}C \in \Gamma^x(R^x) \ \land \ (t_0,\kappa_0) = initSettings(G) \ \land \ \langle C,\mathbb{B} \rangle_{\kappa_0} \not\in \Gamma^x(R^x) \}$$

$$D := \{(\kappa,n) \mid \langle C,\mathbb{B} \rangle_{\kappa} \in \Gamma^x(R^x) \ \land \ n = size(C) \}$$

$$\mathcal{A}^-_{\mathcal{B}} := \{\kappa \mid \langle \vec{A} : D \Rightarrow C,\mathbb{B} \rangle_{\kappa} \in \Gamma^x(R^x) \ \land \ A \in \vec{A} \ \land \ A[\kappa \rhd D] \not\in \Gamma^x(R^x) \}$$

$$\mathcal{S}^-_{reveal} := \{(\kappa,s) \mid \langle \vec{A} : D \Rightarrow C,\mathbb{B} \rangle_{\kappa} \in \Gamma^x(R^x) \ \land \ D = reveal \ \vec{s} \ if \ p \ then \ C' \ \land \ s \in \vec{s}$$

$$\land \ A : s\#N \not\in \Gamma^x(R^x) \}$$

where size is defined as

$$\begin{aligned} \textit{size}(\ \vec{_} : \textit{split}\ \vec{\mathbf{B}} \rightarrow \vec{C} \ \Leftrightarrow C) &= \sum_{C' \in \vec{C}} \textit{size}(C') + \textit{size}(C) + 1 \\ \textit{size}(\ _ : D \ \Leftrightarrow C) &= \textit{size}(D) + \textit{size}(C) + 1 \\ \textit{size}(\ _) &= 1 \end{aligned}$$

Lemma 22 (Decreasing BitML^x finalization distance). Let A be an honest participant with an eager strategy Σ_A^x and K a set of contracts such that Σ_A^x is K-bounded and let R^x (with $\Sigma_A^x \vdash R^x$) be a corresponding run. Then either

- R^x is liquidated i.e. $\not\exists \kappa \in \mathcal{K}. \langle C, \mathbf{B} \rangle_{\kappa} \in \Gamma^x(\dot{R}^x)$
- or there is $\alpha \in \Sigma_A^x(R^x)$ and $\|(R^x, \Sigma_A^x)\| >_{lex} \|(R^x \xrightarrow{\alpha}, \Sigma_A^x)\|$

where $>_{lex}$ denotes the lexicographical ordering on the tuple-output of $\|(\cdot)\|$ with

- $\bullet \ \mathcal{K}^- \leq \dot{\mathcal{K}^-} : \Leftrightarrow \mathcal{K}^- \subseteq \dot{\mathcal{K}^-}$
- $C_{adv}^- \le C_{adv}^- : \Leftrightarrow C_{adv}^- \subseteq C_{adv}^-$
- $\mathcal{A}_{adv}^- \leq \mathcal{A}_{adv}^- : \Leftrightarrow \mathcal{A}_{adv}^- \subseteq \mathcal{A}_{adv}^-$
- $\delta_{adv} \leq \delta_{adv} :\Leftrightarrow \delta_{adv} \subseteq \delta_{adv}$
- $D \le \dot{D} : \Leftrightarrow \forall (\kappa, n) \in D : \exists n'\kappa' : (\kappa', n') \in \dot{D} \land n \le n' \land \kappa' \in desc(\kappa)$
- $\bullet \ \mathcal{A}_{\mathcal{B}}^{-} \leq \dot{\mathcal{A}_{\mathcal{B}}^{-}} : \Leftrightarrow \mathcal{A}_{\mathcal{B}}^{-} \subseteq \dot{\mathcal{A}_{\mathcal{B}}^{-}}$
- $\mathcal{A}_{\mathcal{B}} \leq \mathcal{A}_{\mathcal{B}} : \Leftrightarrow \mathcal{A}_{\mathcal{B}} \subseteq \mathcal{A}_{\mathcal{B}}$ $\mathcal{S}_{reveal}^{-} \leq \mathcal{S}_{reveal}^{-} : \Leftrightarrow \mathcal{S}_{reveal}^{-} \subseteq \mathcal{S}_{reveal}^{-}$

Proof. Assume R^x is not liquidated, then there exists $\kappa \in \mathcal{K}$ such that $\langle C, \mathbf{B} \rangle_{\kappa} \in \Gamma^x(R^x)$ By the non-blocking design of BitML^x contracts and eager strategies we get that

$$\forall \langle C, \mathbf{B} \rangle_{\kappa} \in \Gamma^{x}(R^{x}). \exists \alpha'. \ \alpha'(\kappa) \in \Sigma^{x}(R^{x})$$

The decreasing finalization distance of this strategy step α' is proven by CA on possible actions. Note that only contracts in Kcan be advertised, committed, signed, etc. This bound is used in the finalization distance as limit. Eagerness is used to ensure progress when revealing secrets and authorizing contracts.

Lemma 23 (Liquidity of eager strategies). Let A be an honest participant and Σ_A^x be an eager strategy of A bounded by K. Then Σ_A^x is strongly liquid.

Proof. Let R^x be a $BitML^x$ run such that $\Sigma_A^x \vdash R^x$.

We are constructing an extension \hat{R}^x with actions that are decreasing the finalization distance of $\|(R^x, \Sigma_A^x)\|$ according to Lemma 22. This extension is finite since the lower bound of $\|(\cdot)\|$ is 0. By definition of Lemma 22, the last configuration is liquidated.

C. Intermediate Liquidity

We will extend the notion of liquidity to the intermediate semantics:

Definition 31 (Liquidated runs). Let K be a set of contracts and $R^{||}$ an intermediate run. Then $R^{||}$ is liquidated if for all $(C, \pi)^{\mathbb{B}}_{\kappa} \in \Gamma^{||}(R^{||})$ one of the following holds

```
∃B: π[status] = Assigned(B)
∃B: π[status] = Slashed(B)
∃B: π[status] = CompensatedFrom(B)
∃B: π[status] = Stip-Refunded(B)
∃B: π[status] = Stip-Slashed(B)
∃B: π[status] = Stip-Compensation(B)
```

We define Liquidated-Status to be the set that includes every liquidated status.

Definition 32 (Strong Intermediate Liquidity). Let A be an honest participant with strategy $\Sigma_A^{||}$. Then $\Sigma_A^{||}$ is strongly liquid if for all runs $R^{||}$ there exists an extension $\hat{R}^{||} = R^{||} \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$ such that

```
1) \forall i \in 1 \dots n : \alpha_i \in \Sigma_A^{||}(R^{||} \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{i-1}})
2) R^{||} is liquidated
```

Note that the second condition accounts for the fact that the intermediate semantics does not delete contracts but changes their status to indicate that they have been executed until finalization.

We first define the liquidation distance of an intermediate run $R^{||}$. The liquidation distance is a measure of how close an intermediate run is to liquidation. Intuitively, an intermediate run will always be liquidated because its contracts (on all chains) will either (1) terminate regularly or (2) will be compensated.

To ensure liquidation, we need to consider the honest user strategy $\Sigma_A^{||}$ to be bounded by a set of contracts \mathcal{K} , meaning that $\Sigma_A^{||}$ only proposes contracts from \mathcal{K} and at most once. Otherwise, the strategy could continue advertising new contracts without ever agreeing on time to pass.

This property should be inherited from the underlying Σ_{\perp}^{x} . For this reason,

Definition 33 (Liquidation Distance). Let A be an honest participant with strategy $\Sigma_A^{||}$ and K a set of contracts such that $\Sigma_A^{||}$ is K-bounded. The liquidation distance of an intermediate run $R^{||}$ w.r.t. to $\Sigma_A^{||}$ (written $\|(R^{||}, \Sigma_A^{||})\|$) is formally defined as follows:

$$\|(R^{||}, \Sigma_{\mathbf{A}}^{||})\| := (\mathcal{K}^{-}, \mathcal{C}_{\mathbf{adv}}^{-}, \mathcal{A}_{\mathbf{adv}}^{-}, \delta_{\mathbf{adv}}^{-}, \mathcal{S}_{\mathbf{stip}}^{-}, \delta_{\mathcal{B}}^{\mathbf{stip-status}}, D_{\mathcal{B}}^{\mathbf{stip-phase}}, D_{ES}, \mathcal{S}_{ES}^{-}, D_{\mathcal{B}}^{\mathbf{sync}}, \mathcal{A}_{\mathcal{B}}^{-}, \mathcal{S}_{\mathbf{reveal}}^{-}, \delta_{\mathcal{B}}^{\mathbf{status}}, D_{\mathcal{B}}^{\mathbf{round}})$$

where

```
\mathcal{K}^- := \mathcal{K}/contracts_{adv}(R^{||})
                                 \mathcal{C}^-_{\mathsf{adv}} := \{\kappa_0 \mid \{G\}^{\mathbb{B}} C \in \Gamma^{||}(R^{||}) \land (t_0, \kappa_0) = initSettings(G) \land \mathbf{A} : [\# \triangleright \{G\}^{\mathbb{B}} C \not\in \Gamma^{||}(R^{||})\}
                             \mathcal{A}_{\mathsf{adv}}^- := \{ (\kappa_0, \mathbb{B}) \mid \{G\}^{\mathbb{B}} C \in \Gamma^{||}(R^{||}) \ \land \ (t_0, \kappa_0) = initSettings(G) \ \land \ \mathbf{A} : ! \, \mathbf{B} \, @ \, \vec{x} \in G \}
                                                                                              \wedge \vec{x}[\mathbb{B}] = x \wedge A : [x \triangleright \{G\}^{\mathbb{B}}C] \notin \Gamma^{||}(R^{||})\}
                                   \delta_{\textit{adv}} := \{ (\kappa_0, \mathbb{B}) \mid \{G\}^{\mathbb{B}'} C \in \Gamma^{||}(R^{||}) \ \land \ (t_0, \kappa_0) = initSettings(G) \ \land \ \langle C, \pi \rangle_{\kappa_0}^{\mathbb{B}} \not\in \Gamma^{||}(R^{||}) \}
                                \mathcal{S}^-_{\textit{stip}} := \{ \kappa_0 \mid \langle C, \pi \rangle_{\kappa_0}^{\mathbb{B}} \in \Gamma^{||}(R^{||}) \ \land \ is_{\kappa_0}^{\textbf{A}} \in \pi[initSecrets] \ \land \ \textbf{A} : is_{\kappa_0}^{\textbf{A}} \not\in \Gamma^{||}(R^{||}) \}
     \delta_{\mathcal{B}}^{\textit{stip-status}} := \{ (\kappa_0, \underline{\mathbb{B}}, \textit{status}) \mid \langle C, \pi \rangle_{\kappa_0}^{\underline{\mathbb{B}}} \in \Gamma^{||}(R^{||}) \ \land \ \textit{status} = \pi[\textit{status}] \}
D_{\mathcal{B}}^{\textit{stip-phase}} := \{ (\kappa_0, \mathbb{B}, 1-p) \mid \langle C, \pi \rangle_{\kappa_0}^{\mathbb{B}} \in \Gamma^{||}(R^{||}) \land p = ? \}
                              D_{ES} := \{ (\kappa, n) \mid \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma^{||}(R^{||}) \land n = size(C) \}
                              \mathcal{S}_{ES}^- := \{\kappa \mid \langle C, \pi \rangle_\kappa^{\mathrm{B}} \in \Gamma^{||}(R^{||}) \ \land \ \textit{$A$}: \ s_\kappa^{\textit{$A$}} \not \in \Gamma^{||}(R^{||})\}
                        D_{\mathcal{B}}^{\mathit{sync}} := \{ (\kappa, \mathbb{B}, n) \mid \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma^{||}(R^{||}) \ \land \ n = \max\{ |\kappa^*| : \langle C', \pi \rangle_{\kappa'}^{\mathbb{B}'} \in \Gamma^{||}(R^{||}) \ \land \kappa' = \kappa \cdot \kappa^* \} \}
                                    \mathcal{A}_{\mathcal{B}}^{-} := \{ (\kappa, \mathbb{B}) \mid \langle \vec{A} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma^{||}(R^{||}) \land A \in \vec{A} \land A[(\kappa, \mathbb{B}) \triangleright D] \notin \Gamma^{||}(R^{||}) \}
                      \mathcal{S}_{\textit{reveal}}^{-} := \{ (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \textit{reveal $\vec{s}$ if $p$ then $C'$} \ \land \ s \in \vec{s} = (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \text{reveal $\vec{s}$ if $p$ then $C'$} \ \land \ s \in \vec{s} = (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \text{reveal $\vec{s}$ if $p$ then $C'$} \ \land \ s \in \vec{s} = (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}) \ \land \ \ D = \kappa \mid (\kappa, s) \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} = \kappa \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} = \kappa \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} = \kappa \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} = \kappa \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} = \kappa \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} = \kappa \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} = \kappa \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} = \kappa \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}^{\mathbf{B}} = \kappa \mid \langle \vec{\pmb{A}} : D \Rightarrow C, \pi \rangle_{\kappa}
                                                                                            \wedge A: s \# N \not\in \Gamma^{||}(R^{||})
                       \delta_{\mathcal{R}}^{\textit{status}} := \{ (\kappa, \mathbb{B}, \textit{status}) \mid \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma^{||}(R^{||}) \ \land \ \textit{status} = \pi[\textit{status}] \}
                    D_{\mathcal{B}}^{round} := \{(\kappa, \mathbb{B}, |\kappa| - r, 1 - p) \mid \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma^{||}(R^{||}) \land (r, p) = roundStatus(\kappa, R^{||})\}
```

where size is defined as

$$\begin{aligned} \textit{size}(\ \vec{_} : \textit{split}\ \vec{\mathsf{B}} \to \vec{C} \ \Leftrightarrow C) &= \sum_{C' \in \vec{C}} \textit{size}(C') + \textit{size}(C) + 1 \\ \textit{size}(\ \underline{_} : D \ \Leftrightarrow C) &= \textit{size}(D) + \textit{size}(C) + 1 \\ \textit{size}(\ \underline{_} \) &= 1 \end{aligned}$$

Intuitively, the different components forming the liquidation distance have the following meaning:

- \mathcal{K}^- denotes the set of contracts κ_0 from \mathcal{K} which have not been advertised yet.
- C_{adv}^- denotes the set of advertised contracts κ_0 that are still lacking commitments (from user A).
- \mathcal{A}_{adv}^- denotes the set of advertised contracts and blockchains (κ_0, \mathbb{B}) that are still lacking authorizations (from user A). Note that authorizations are carried out per blockchain \mathbb{B} .
- δ_{adv} denotes the set of advertised contracts and blockchains (κ_0, \mathbb{B}) that have not been published yet. Note that contracts κ_0 need to be published individually on each blockchain.
- S_{stip}^- denotes the set of published contracts κ_0 that are still lacking the init secrets (from user A). Note that there is a single init secret per contract κ_0 and user that honest A will only release once the contract κ_0 has been published on all chains.
- $\delta_{\mathcal{B}}^{\text{stip-status}}$ denotes the set of contracts, blockchain with their corresponding stipulation status $(\kappa_0, \mathbb{B}, status)$. Note that a contract κ_0 might be in different stages of stipulation (indicated by status) on the different blockchains
- $D_{\mathcal{B}}^{\text{stip-phase}}$ denotes the set of contracts with their corresponding stipulation phase (κ_0, p) .
- D_{ES} denotes the set of all running contracts with their finalization distance (κ, n) , which is given by the depth of the contract. Note that since contracts are non-recursive, the maximal number of execution steps is bounded by the contract structure.
- S_{ES}^- denotes the set of all running contracts κ that are missing a step secret from A.
- $D_{\mathcal{B}}^{\text{sync}}$ denotes the set of running contracts, blockchains and their synchronization distance (κ, \mathbb{B}, n) . The synchronization distance n denotes how far the contract κ on blockchain \mathbb{B} is lacking behind its furthest descendant (on any other chain).
- $\mathcal{A}_{\mathcal{B}}^-$ denotes the set of running (priority choice) contracts and blockchains (κ, \mathbb{B}) that are still lacking authorizations from A.
- $\delta_{\mathcal{B}}^{\text{status}}$ denotes the set of running contracts, blockchains and their execution stats $(\kappa, \mathbb{B}, status)$. Note that a contract κ may be in different stages of execution on the different blockchains.
- S_{reveal}^- denotes the set of running contracts their secrets that need to be revealed (κ, s) .

• $D_{\mathcal{B}}^{\text{round}}$ denotes the set of running contracts, blockchains and the corresponding execution round and phase finalization distances $(\kappa, someBlockchain, r_{\delta}, p_{\delta})$. Here, r_{δ} denotes the number of rounds that need to pass until execution of κ will be latest enforced and p_{δ} denotes the phase within the enforcement round.

Lemma 24 (Non-blocking intermediate contracts). Given an intermediate run $R^{||}$, we can proof that every non-liquidated intermediate contract has at least a single active move:

$$\forall \langle C, \pi \rangle_{\kappa}^{\mathbf{B}} \in \Gamma^{||}(R^{||}).status[\pi] \not\in \mathit{Liquidated-Status} \implies (\exists \delta \alpha \Gamma^{||}. \ R^{||} \xrightarrow{\delta \alpha(\kappa)} \Gamma^{||})$$

SSkip] and [——Abort].

Lemma 25 (Decreasing finalization distance). Let A be an honest participant with an eager strategy Σ_A^x and K a set of contracts such that Σ_A^x is K-bounded. Let $\Sigma_A^{||} = \mathcal{S}(\Sigma_A^x)$ be the corresponding intermediate strategy and let R^x (with $\Sigma_A^x \vdash R^x$) and $R^{||}$ (with $\Sigma_A^{||} \vdash R^{||}$) such that $R^x \sim_A R^{||}$. Then either

- R|| is liquidated
- or there is $\alpha \in \Sigma_A^{||}(R^{||})$ and $\|(R^{||}, \Sigma_A^{||})\| >_{lex} \|(R^{||} \xrightarrow{\alpha}, \Sigma_A^{||})\|$

where $>_{lex}$ denotes the lexicographical ordering on the tuple-output of $\|(\cdot)\|$ with

- $\mathcal{K}^- \leq \dot{\mathcal{K}^-} : \Leftrightarrow \mathcal{K}^- \subseteq \dot{\mathcal{K}^-}$
- $\bullet \ \ \mathcal{C}^-_{\textit{adv}} \leq \mathcal{C}^-_{\textit{adv}} : \Leftrightarrow \mathcal{C}^-_{\textit{adv}} \subseteq \mathcal{C}^-_{\textit{adv}}$
- $\begin{array}{l} \bullet \;\; \mathcal{A}_{\textit{adv}}^{-} \leq \mathcal{A}_{\textit{adv}}^{-} : \Leftrightarrow \mathcal{A}_{\textit{adv}}^{-} \subseteq \mathcal{A}_{\textit{adv}}^{-} \\ \bullet \;\; \delta_{\textit{adv}} \leq \delta_{\textit{adv}} : \Leftrightarrow \delta_{\textit{adv}} \subseteq \delta_{\textit{adv}} \end{array}$

- $S_{stip}^{\bullet} \leq S_{stip}^{\bullet} :\Leftrightarrow S_{stip}^{\bullet} \subseteq S_{stip}^{\bullet}$ $S_{stip}^{\bullet} \leq S_{stip}^{\bullet} :\Leftrightarrow S_{stip}^{\bullet} \subseteq S_{stip}^{\bullet}$ $\delta_{\mathcal{B}}^{stip\text{-status}} \leq \delta_{\mathcal{B}}^{stip\text{-status}} :\Leftrightarrow \forall (\kappa_0, \mathbb{B}, status) \in \delta_{\mathcal{B}}^{stip\text{-status}} : \exists status' : (\kappa_0, \mathbb{B}, status') \in \delta_{\mathcal{B}}^{stip\text{-status}} \land status \leq_{stip\text{-status}} status'$ $D_{\mathcal{B}}^{stip\text{-phase}} \leq D_{\mathcal{B}}^{stip\text{-phase}} :\Leftrightarrow \forall (\kappa_0, \mathbb{B}, n) \in D_{\mathcal{B}}^{stip\text{-phase}} : \exists n' : (\kappa_0, \mathbb{B}, n') \in D_{\mathcal{B}}^{stip\text{-phase}} \land n \leq n'$ $D_{ES} \leq D_{ES} :\Leftrightarrow \forall (\kappa, n) \in D_{ES} : \exists n'\kappa' : (\kappa', n') \in D_{ES} \land n \leq n' \land \kappa' \in desc(\kappa)$

- $\bullet \ \mathcal{S}_{ES}^{-} \leq \dot{\mathcal{S}_{ES}^{-}} : \Leftrightarrow \mathcal{S}_{ES}^{-} \subseteq \dot{\mathcal{S}_{ES}^{-}}$ $\bullet \ D_{\mathcal{B}}^{\mathit{sync}} \leq D_{\mathcal{B}}^{\mathit{sync}} : \Leftrightarrow \forall (\kappa, \mathbb{B}, n) \in D_{\mathcal{B}}^{\mathit{sync}} : \exists n' \kappa' : (\kappa', \mathbb{B}, n') \in D_{\mathcal{B}}^{\mathit{sync}} \ \land \ n \leq n' \ \land \ \kappa' \in desc(\kappa)$
- $\mathcal{A}_{\mathcal{B}}^{-} \leq \mathcal{A}_{\mathcal{B}}^{-} : \Leftrightarrow \mathcal{A}_{\mathcal{B}}^{-} \subseteq \mathcal{A}_{\mathcal{B}}^{-}$

- $S_{\mathcal{B}}^{\mathsf{B}} \subseteq \mathcal{G}_{\mathcal{B}}^{\mathsf{C}} : \mathcal{G}_{\mathcal{B}}^{\mathsf{B}} \subseteq \mathcal{G}_{\mathcal{B}}^{\mathsf{B}}$ $S_{\mathsf{reveal}}^{\mathsf{C}} \subseteq S_{\mathsf{reveal}}^{\mathsf{C}} : \Leftrightarrow S_{\mathsf{reveal}}^{\mathsf{C}} \subseteq S_{\mathsf{reveal}}^{\mathsf{C}}$ $\delta_{\mathcal{B}}^{\mathsf{status}} \subseteq \delta_{\mathcal{B}}^{\mathsf{status}} : \Leftrightarrow \forall (\kappa, \mathbb{B}, \mathsf{status}) \in \delta_{\mathcal{B}}^{\mathsf{status}} : \exists \mathsf{status}' : (\kappa, \mathbb{B}, \mathsf{status}') \in \delta_{\mathcal{B}}^{\mathsf{status}} \land \mathsf{status} \leq s_{\mathsf{status}} \mathsf{status}'$ $D_{\mathcal{B}}^{\mathsf{round}} \subseteq D_{\mathcal{B}}^{\mathsf{round}} : \Leftrightarrow \forall (\kappa, \mathbb{B}, r_d, p_d) \in D_{\mathcal{B}}^{\mathsf{round}} : \exists r_d' \ p_d' : (\kappa_0, \mathbb{B}, r_d, p_d') \in D_{\mathcal{B}}^{\mathsf{round}} \land (r_d, p_d) \leq l_{\mathsf{ex}} (r_d', p_d')$

and

```
\leq_{status} := \{(CompensatedFrom(A), Slashed(A)), \}
           (Slashed(A), Right),
           (Right, Choice),
           (Left, Choice),
           (Assigned(A), Left)*
```

and

$$\leq_{\textit{stip-status}} := \{(\textit{Stip-Compensation}(A), \textit{Stip-Slashed}(A)), \\ (\textit{Stip-Slashed}(A), \textit{Stip-Right}), \\ (\textit{Stip-Right}, \textit{Stip-Choice}), \\ (\textit{Choice}, \textit{Stip-Choice}), \\ (\textit{Stip-Refunded}(A), \textit{Stip-Right})\}^*$$

(With R^* we denote the reflexive and transitive closure of a relation R)

Proof. By definition of intermediate strategy compilation, we know that

$$\exists \alpha' \in \Sigma_{\mathbf{A}}^{||}(R^{||}) \land (\forall \Gamma^{||} : R^{||} \xrightarrow{\alpha'} \Gamma^{||} \Rightarrow \Gamma^{||} \neq \Gamma^{||}(R^{||}))$$

The progress of configuration is natural for timed moves. a For stipulation commitments and authorizations progress is derived from BitML^x strategy eagerness and run coherence.

It is left to be shown that each move decreases the finalization distance. Note that only contracts in K can be advertised, committed, signed, etc. This bound is used in the finalization distance as limit. Eagerness is used to ensure progress when revealing secrets and authorizing contracts.

CA on possible moves of $\Sigma_A^{||}$:

- Advertise: $advertise(\{G\}C)$: \mathcal{K}^- decreases
- Authorize Commit: $commit(A, \{G\}C)$: only C_{adv}^- decreases. K-Eagerness ensures uniqueness of authorization.
- Authorize Init: $authInit(A, \{G\}^{\mathbb{B}}C)$: only \mathcal{A}_{adv}^{-} decreases.
- Publish: $publish(\{G\}^{\mathbb{B}}C)$: δ_{adv} decreases and more significant components stall
- Double Spend: $doubleSpend(A, \{G\}^{\mathbb{B}}C)$

All of the following moves emitted by Σ_A^{\parallel} do not increase/decrease $\mathcal{K}^-, \mathcal{C}_{adv}^-, \mathcal{A}_{adv}^-$ and δ_{adv} :

- Reveal Init Step Secret: $A:is^{A}_{\kappa}$: only decreases $\mathcal{S}^{-}_{\mathsf{stip}}$
- Init: $init(\kappa, \mathbb{B})$: only $\delta_{\mathcal{B}}^{\text{stip-status}}$ decreases from **Stip-Choice** to **Choice**
- Stip Skip: $from \ t: \ sskip(\kappa, \mathbb{B})$: only $\delta^{\mathsf{stip}\text{-status}}_{\mathcal{B}}$ decreases from **Stip-Choice** to **Stip-Right**
- Abort: $from \ t: \ abort(\kappa, \mathbb{B})$: only $\delta^{\text{stip-status}}_{\mathcal{B}}$ decreases from **Stip-Right** to **Stip-Refunded** (A_i)
- Intro Stip Compensate: $sslash(\kappa_0, \mathbb{B}, A)$: only $\delta_{\mathcal{B}}^{\text{stip-status}}$ decreases from **Stip-Right** to **Stip-Slashed**(A)
- Stip Compensate: $scompensate(\kappa_0, \mathbb{B}, A)$: only $\delta_{\mathcal{B}}^{\text{stip-status}}$ **Stip-Compensation**(*A*)

In addition, all of the following moves do not increase/decrease $\mathcal{S}_{\mathsf{stip}}^-, \delta_{\mathcal{B}}^{\mathsf{stip-status}}$ and $D_{\mathcal{B}}^{\mathsf{stip-phase}}$:

- Secret Reveal: A:a: only $\mathcal{S}^-_{\text{reveal}}$ decreases
 Step Secret Reveal: $A:s_{\kappa}^A$: only \mathcal{S}^-_{ES} decreases
 Intro Left: $ileft(\kappa,\mathbb{B},x)$: only $\delta^{\text{status}}_{\mathcal{B}}$ decreases from **Choice** to **Left**
- Reveal: $reveal(\kappa, \mathbb{B}, x)$: If ES frontier is moved with this action, D_{ES} is decreased. Otherwise $D_{\mathcal{B}}^{\mathsf{sync}}$ decreases but D_{ES} and S_{ES}^- remains the same.
- Authorize: $A:(\kappa,\mathbb{B},x)$: only $\mathcal{A}_{\mathcal{B}}^-$ decreases. Eagerness ensures that authorization is unique. Intro Skip: $from\ t:\ skip(\kappa,\mathbb{B},x)$: only $\delta_{\mathcal{B}}^{\text{status}}$ decreases from **Choice** to **Right**
- Elim Skip: $from \ t : \ right(\kappa, \mathbb{B}, x)$: If ES frontier is moved with this action, D_{ES} is decreased. Otherwise $D_{\mathcal{B}}^{\mathsf{sync}}$ decreases but D_{ES} and \mathcal{S}_{ES}^- remains the same.

 • Intro Compensation: $slash(\kappa,\mathbb{B},x,\pmb{A})$: only $\delta_{\mathcal{B}}^{\text{status}}$ decreases from **Right** to **Slashed**(\pmb{A})
- Elim Compensation: $compensate(\kappa,\mathbb{B},x,A)$: only $\delta^{\text{status}}_{\mathcal{B}}$ decreases from Slashed(A) to CompensatedFrom(A)• Guarded Withdraw: $dwithdraw(\kappa,\mathbb{B},x)$: only $\delta^{\text{status}}_{\mathcal{B}}$ decreases from Left to Assigned(A)
- Split: $split(\kappa,\mathbb{B},x)$: If ES frontier is moved with this action, D_{ES} is decreased. Otherwise $D_{\mathcal{B}}^{\mathsf{sync}}$ decreases but D_{ES} and \mathcal{S}_{ES}^{-} remains the same.
- Withdraw: $cwithdraw(\kappa, \mathbb{B}, x)$: only $\delta_{\mathcal{B}}^{\text{status}}$ decreases from **Choice** to **Assigned**(\underline{A})

Time delays δ do not change the untimed configurations. By assumption, the honest user strategy $\Sigma_A^{||}$ proposes only delays that changes the phase/round of at least a single contract.

Let $\hat{\kappa}$ be the identifier of the contract with the next round/phase deadline in \mathcal{K} and $\hat{\delta}$ be the delay to be scheduled by an honest user strategy Σ_A^{\parallel} that wants to reach the deadline without scheduling any untimed moves before. If κ is not yet initialised, $D_{\mathcal{B}}^{\text{stip-phase}}$ is decreased by this $\hat{\delta}$ move by definition of stip-phase. Otherwise, $D_{\mathcal{B}}^{\text{round}}$ is decreased by the definition of roundStatus($\hat{\kappa}, R^{||}$).

Lemma 26 (Liquidity of intermediate strategies). Let A be an honest participant with an eager strategy Σ_A^x and K a set of contracts such that Σ_A^x is K-bounded. Let $\Sigma_A^{||} = \mathcal{S}(\Sigma_A^x)$ be the corresponding intermediate strategy. Then $\Sigma_A^{||}$ is strongly

Proof. The definition of strongly liquid requires the construction of a finite trace extension $\hat{R}^{||}$ according to $\hat{\Sigma}_{4}^{||}$ until $\hat{R}^{||}$ is liquidated.

We construct the extension $R^{||}$ with actions that are decreasing the finalization distance of $\|(R^{||}, \Sigma_A^{||})\|$ using Lemma 25. This extension is finite since the lower bound of the finalization distance $\|(\cdot)\|$ is 0. By definition of Lemma 25, the last configuration is liquidated.

APPENDIX L BitML* SECURITY

A. Payouts and Contract Funds

We first introduce some auxiliary infrastructure and lemmas that help us to reason about inputs.

First, we define the intermediate inputs per contract.

Definition 34 (Intermediate inputs per contract). Let $R^{||}$ be an intermediate semantics run and \mathbb{B} be a blockchain. We define the intermediate inputs of A in $R^{||}$ and blockchain \mathbb{B} (written inputs $_{A;\kappa_0}^{\mathbb{B}}(R^{||})$) as follows

$$\sum_{\{v:A:!\,v\mathbb{B}\;@\,x\in G^{\mathbb{B}}\land publish(\{G\}^{\mathbb{B}}\,C)\in R^{||}\land (t_0,\kappa_0)=initSettings(G)\}}\sigma(v:A:!\,v\mathbb{B}\;@\,x\in G^{\mathbb{B}}\land publish(\{G\}^{\mathbb{B}}\,C)\in R^{||}\land (t_0,\kappa_0)=initSettings(G)\}$$

Similarly, we define the total intermediate funds per contracts:

Definition 35 (Intermediate total funds per contract). Let $R^{||}$ be an intermediate semantics run and \mathbb{B} be a blockchain. We define the contract funds in $R^{||}$ (written totalFunds $_{\kappa_0}^{\mathbb{B}}(R^{||})$) as follows

$$\sum_{\{\pi: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}} \ \land \ \kappa \in desc(\kappa_{0}, R^{||})\}} \pi.balance$$

Intermediate runs preserve inputs per contract:

Lemma 27 (Intermediate money preservation per contract). Let $R^{||}$ be an intermediate semantics run and $\kappa_0 \in R^{||}$. Then, for every blockchain \mathbb{B} , the following equation holds

$$\mathit{totalFunds}^{\mathrm{B}}_{\kappa_0}(R^{||}) = \sum_{A \in P} \mathit{inputs}^{\mathrm{B}}_{A:\kappa_0}(R^{||})$$

Further, we can show the following:

Lemma 28 (Intialized eventual synchronicty implies publication). Let $R^{||}$ be an intermediate semantics run and $\kappa_0 \in R^{||}$ and $stipStatus^{es}(R^{||}, \kappa_0) = Initialized$. Then

$$\forall \mathbb{B} : publish(\{G^{\mathbb{B}}\}C) \in R^{||}$$

Proof. Same proof strategy as in the general intermediate money preservation lemma (Lemma 18).

Lemma 29 (Liquidated Run Security). Let A be an honest participant with strategy Σ_A^x and $\Sigma_A^{||} = \mathcal{S}(\Sigma_A^x)$ its corresponding intermediate strategy. Let R^x and $R^{||}$ be such that $R^x \sim_A R^{||}$ and $R^{||}$ being liquidated. Then

$$\forall \mathbb{B} \in \mathcal{B}: \textit{payout}^{\mathbb{B}}_{A}(R^{||}) - \textit{inputs}^{\mathbb{B}}_{A}(R^{||}) \geq \textit{payout}^{\mathbb{B}}_{A}(R^{x}) + \textit{contractFunds}^{\mathbb{B}}(R^{x}) - \textit{inputs}^{\mathbb{B}}_{A}(R^{x})$$

Intuitively, this statement says that in a fully liquidated intermediate run $R^{||}$, the honest user earned at least as much as in the coherent $BitML^x$ R^x plus the money that is still locked in contracts of R^x . The intuition behind this statement is that contracts in R^x can only be still present in R^x because they have been fully compensated.

Proof. Let $\mathbb{B} \in \mathcal{B}$. From the definition of \sim_A we know that

$$ES = \mathsf{maxFrontier}(R^x) = \bigsqcup_{\mathbb{B} \in \mathcal{B}} \mathsf{maxFrontier}(R^{||}, \underline{\mathbb{B}})$$

Consequently, we can conclude that

$$\forall \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}} \land \kappa \in ES \Rightarrow \exists O : \langle O, \mathbf{B} \rangle_{\kappa} \in \Gamma_{R^{x}} \land \mathbf{B}[\mathbb{B}] = \pi.balance \tag{1}$$

And consequently for all $K \subseteq ES$ such that $\forall \kappa \in K : \exists \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{|||}}$ we have that

$$\sum_{\kappa \in K: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}} \pi.balance = \sum_{\kappa \in K: \langle O, \mathbb{B} \rangle_{\kappa} \in \Gamma_{R^{x}}} \mathbb{B}[\mathbb{B}]$$
 (2)

From coherence, we know that

$$orall \kappa_0 \in R^{||}: extstyle ex$$

With that we can show that

$$K_0 := \{(\kappa_0, A, \mathbb{B}, v) \mid init(\{G\}C) \in R^x \land (t_0, \kappa_0) = initSettings(G) \\ \land A : ! \ B @ \vec{x} \in G \land v = B[\mathbb{B}] \}$$

$$= \{(\kappa_0, A, \mathbb{B}, v) \mid stipStatus^x(R^x, \kappa_0) = Initialized() \land \{G\}C \in R^x \land (t_0, \kappa_0) = initSettings(G) \\ \land A : ! \ B @ \vec{x} \in G \land v = B[\mathbb{B}] \}$$

$$= \{(\kappa_0, A, \mathbb{B}, v) \mid stipStatus^{es}(R^{||}, \kappa_0) = Initialized() \land \{G\}C \in R^{||} \land (t_0, \kappa_0) = initSettings(G) \\ \land A : ! \ B @ \vec{x} \in G \land v = B[\mathbb{B}] \}$$

$$= \{(\kappa_0, A, \mathbb{B}, v) \mid publish(\{G^{\mathbb{B}}\}C) \in R^{||} \land stipStatus^{es}(R^{||}, \kappa_0) = Initialized() \\ \land (t_0, \kappa_0) = initSettings(G) \land A : ! \ B @ \vec{x} \in G \land v = B[\mathbb{B}] \}$$

The last equation holds due to Lemma 28.

As a consequence, we have that

$$\sum_{\pmb{A}\in P}\mathsf{inputs}^{\mathbb{B}}_{\pmb{A}}(R^x) = \sum_{\pmb{A}\in P, \kappa_0\in\{\kappa_0:\kappa_0\in R^{||}\ \land\ \mathsf{stipStatus}^{es}(R^{||},\kappa_0) = \mathit{Initialized}()\}}\mathsf{inputs}^{\mathbb{B}}_{\pmb{A}:\kappa_0}(R^{||}) \tag{3}$$

Since $R^{||}$ is liquidated we know that all contracts are either initialized or aborted, so we get that:

$$\sum_{A \in P} \mathsf{inputs}_{A}^{\mathbb{B}}(R^{||}) = \sum_{\substack{A \in P, \kappa_0 \in \{\kappa_0 : \kappa_0 \in R^{||} \ \land \ \mathsf{stipStatus}^{es}(R^{||}, \kappa_0) = \mathit{Initialized}()\}}} \mathsf{inputs}_{A : \kappa_0}^{\mathbb{B}}(R^{||})$$

$$+ \sum_{\substack{A \in P, \kappa_0 \in \{\kappa_0 : \kappa_0 \in R^{||} \ \land \ \mathsf{stipStatus}^{es}(R^{||}, \kappa_0) = \mathit{Aborted}\}}} \mathsf{inputs}_{A : \kappa_0}^{\mathbb{B}}(R^{||})$$

$$(5)$$

Finally, we can show using Lemma 27 that inputs of the contracts in $\{\kappa_0 : \kappa_0 \in R^{||} \land \text{ stipStatus}^{es}(R^{||}, \kappa_0) = Aborted\}$ sum up to the corresponding contract funds, which we know must be either refunded or compensated. More precisely, we have that

$$\begin{split} \sum_{\substack{A \in P, \kappa_0 \in \{\kappa_0 : \kappa_0 \in R^{||} \ \land \ \text{stipStatus}^{es}(R^{||}, \kappa_0) = Aborted\}}} & \text{inputs}_{A : \kappa_0}^{\mathbb{B}}(R^{||}) \\ &= \sum_{\{\kappa_0 : \text{stipStatus}^{\mathbb{B}}(R^{||}, \kappa_0) = Refunded\}} & \text{totalFunds}_{\kappa_0}^{\mathbb{B}}(R^{||}) \\ &+ \sum_{\{\kappa_0 : \text{stipStatus}^{\mathbb{B}}(R^{||}, \kappa_0) = Aborted\}} & \text{totalFunds}_{\kappa_0}^{\mathbb{B}}(R^{||}) \end{split}$$

Note that if a contract κ_0 is refunded on one chain, this implies that its eventual synchronicity status is *Aborted*. For conciseness, we will use the following sets in the remainder of the proof:

$$\overline{ES} := \{\kappa : \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}} \land \kappa \notin ES\}$$

$$F_{\mathbb{B}} := \{\kappa : \langle O, \mathbb{B} \rangle_{\kappa} \in \Gamma_{R^{x}} \land \exists \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}\}$$

$$\overline{F_{\mathbb{B}}} := \{\kappa : \langle O, \mathbb{B} \rangle_{\kappa} \in \Gamma_{R^{x}} \land \exists \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}\}$$

$$Abort := \{\kappa_{0} : \text{stipStatus}^{\mathbb{B}}(R^{||}, \kappa_{0}) = Compensated \land \text{stipStatus}^{es}(R^{||}, \kappa_{0}) = Aborted\}$$

$$Refund := \{\kappa_{0} : \text{stipStatus}^{\mathbb{B}}(R^{||}, \kappa_{0}) = Refunded\}$$

$$Ass(A) := \{\kappa : \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}} \land \pi[status] = \mathbf{Assigned}(A)\}$$

$$Comp := \{\kappa : \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}} \land \exists B : B \neq A \land \pi[status] = \mathbf{CompensatedFrom}(B)\}$$

We know that for both x and intermediate runs the total funds and inputs sum up. So consequently, it holds that:

$$\mathsf{totalFunds}^{\mathbb{B}}(R^{||}) - \sum_{\pmb{A} \in P} \mathsf{inputs}^{\mathbb{B}}_{\pmb{A}}(R^{||}) = \sum_{\pmb{A} \in P} \mathsf{payout}^{\mathbb{B}}_{\pmb{A}}(R^x) + \mathsf{contractFunds}^{\mathbb{B}}(R^x) - \sum_{\pmb{A} \in P} \mathsf{inputs}^{\mathbb{B}}_{\pmb{A}}(R^x) \tag{6}$$

Since

$$\mathsf{totalFunds}^{\mathbb{B}}(R^{||}) = \sum_{\kappa \in ES: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R||}} \pi.balance + \sum_{\kappa \in \overline{ES}: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R||}} \pi.balance$$

and

$$\sum_{\pmb{A}\in P}\mathsf{payout}_{\pmb{A}}^{\mathbb{B}}(R^x) + \mathsf{contractFunds}^{\mathbb{B}}(R^x) = \sum_{\kappa\in F_{\mathbb{B}}:\langle O, \mathbb{B}\rangle_{\kappa}\in \Gamma_{R^x}} \mathbf{B}[\mathbb{B}] + \sum_{\kappa\in \overline{F_{\mathbb{B}}}:\langle O, \mathbb{B}\rangle_{\kappa}\in \Gamma_{R^x}} \mathbf{B}[\mathbb{B}]$$

From 1 we know that

$$\sum_{\kappa \in ES: \langle C, \pi \rangle_{\kappa}^{\mathtt{B}} \in \Gamma_{R||}} \pi.balance = \sum_{\kappa \in \mathit{F}_{\mathtt{B}}: \langle O, \mathtt{B} \rangle_{\kappa} \in \Gamma_{R^{x}}} \mathtt{B}\left[\mathbb{B}\right]$$

and consequently, we can conclude from 6 that

$$\sum_{\kappa \in \overline{F_{\mathbb{B}}}: \langle O, \mathbb{B} \rangle_{\kappa} \in \Gamma_{R^x}} \mathbb{B}[\mathbb{B}] = \sum_{\kappa \in \overline{ES}: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}} \pi.balance + \sum_{A \in P} \mathsf{inputs}_{A}^{\mathbb{B}}(R^x) - \sum_{A \in P} \mathsf{inputs}_{A}^{\mathbb{B}}(R^{||})$$

Substituting 3 and 5 we obtain

$$\begin{split} \sum_{\kappa \in \overline{F_{\mathbb{B}}}: \langle O, \mathbb{B} \rangle_{\kappa} \in \Gamma_{R^{x}}} \mathbb{B}\left[\mathbb{B}\right] &= \sum_{\kappa \in \overline{ES}: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}} \pi.balance - \sum_{\kappa_{0} \in \textit{Refund}} \mathsf{totalFunds}_{\kappa_{0}}^{\mathbb{B}}(R^{||}) \\ &- \sum_{\kappa_{0} \in \textit{Abort}} \mathsf{totalFunds}_{\kappa_{0}}^{\mathbb{B}}(R^{||}) \end{split}$$

In particular, we know that all contracts in \overline{ES} have been compensated (since $R^{||}$ is liquidated and contracts that are lacking behind the R^x need to be compensated).

So, we know that:

$$\begin{split} \sum_{\kappa \in \overline{ES} \cap \mathit{Comp}: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}} \pi.\mathit{balance} &= \sum_{\kappa \in \overline{F_{\mathbb{B}}}: \langle O, \mathbb{B} \rangle_{\kappa} \in \Gamma_{R^{x}}} \mathbb{B}\left[\mathbb{B}\right] + \sum_{\kappa_{0} \in \mathit{Refund}} \mathsf{totalFunds}_{\kappa_{0}}^{\mathbb{B}}(R^{||}) \\ &+ \sum_{\kappa_{0} \in \mathit{Abort}} \mathsf{totalFunds}_{\kappa_{0}}^{\mathbb{B}}(R^{||}) \end{split}$$

Using this, we can show our final statement:

$$\begin{split} &\operatorname{payout}_{A}^{\mathbb{B}}(R^{||}) - \operatorname{inputs}_{A}^{\mathbb{B}}(R^{||}) \\ &= \sum_{\kappa \in \operatorname{Ass}(A): \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}} \pi.\operatorname{balance} + \sum_{\kappa \in \operatorname{Comp}: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}} \pi.\operatorname{balance} - \operatorname{inputs}_{A}^{\mathbb{B}}(R^{||}) \\ &= \sum_{\kappa \in \operatorname{Ass}(A) \cap ES: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}} \pi.\operatorname{balance} + \sum_{\kappa \in \operatorname{Comp} \cap ES: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}} \pi.\operatorname{balance} + \sum_{\kappa \in \operatorname{Comp} \cap ES: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}} \pi.\operatorname{balance} + \sum_{\kappa \in \operatorname{Comp} \cap ES: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}} \pi.\operatorname{balance} + \sum_{\kappa \in \operatorname{Comp} \cap ES: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}} \mathbb{B}[\mathbb{B}] \\ &= \sum_{\kappa \in \operatorname{Ass}(A) \cap F_{\mathbb{B}}: \langle O, \mathbb{B} \rangle_{\kappa} \in \Gamma_{R^{||}}} \mathbb{B}[\mathbb{B}] + \sum_{\kappa \in \operatorname{Comp} \cap ES: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}} \pi.\operatorname{balance} + \sum_{\kappa \in F_{\mathbb{B}}: \langle O, \mathbb{B} \rangle_{\kappa} \in \Gamma_{R^{||}}} \mathbb{B}[\mathbb{B}] \\ &+ \sum_{\kappa \in \operatorname{Refind}} \operatorname{totalFunds}_{\kappa_0}^{\mathbb{B}}(R^{||}) + \sum_{\kappa_0 \in \operatorname{Aborr}} \operatorname{totalFunds}_{\kappa_0}^{\mathbb{B}}(R^{||}) - \operatorname{inputs}_{A}^{\mathbb{B}}(R^{||}) \\ &= \operatorname{payout}_{A}^{\mathbb{B}}(R^{\mathbb{B}}) + \operatorname{contractFunds}_{\kappa_0}^{\mathbb{B}}(R^{||}) + \sum_{\kappa \in \operatorname{Comp} \cap ES: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}} \pi.\operatorname{balance} \\ &+ \sum_{\kappa_0 \in \operatorname{Refind}} \operatorname{totalFunds}_{\kappa_0}^{\mathbb{B}}(R^{||}) + \sum_{\kappa_0 \in \operatorname{Aborr}} \operatorname{totalFunds}_{\kappa_0}^{\mathbb{B}}(R^{||}) - \operatorname{inputs}_{A}^{\mathbb{B}}(R^{||}) \end{aligned}$$

Consequently, we only need to show that

$$\begin{split} & \mathsf{inputs}^{\mathbb{B}}_{A}(R^{x}) \geq \mathsf{inputs}^{\mathbb{B}}_{A}(R^{||}) \\ & - \sum_{\kappa \in \mathit{Comp} \cap ES: \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in \Gamma_{R^{||}}} \pi.\mathit{balance} - \sum_{\kappa_{0} \in \mathit{Refund}} \mathsf{totalFunds}^{\mathbb{B}}_{\kappa_{0}}(R^{||}) - \sum_{\kappa_{0} \in \mathit{Abort}} \mathsf{totalFunds}^{\mathbb{B}}_{\kappa_{0}}(R^{||}) \end{split}$$

This immediately follow from 3 and 5.

Lemma 30 (Intermediate Security). Let A be an honest participant and K be a set of contract identifiers. Let Σ_A^x be an eager and deterministic strategy of A bounded by K and $\Sigma_A^{||} = S(\Sigma_A^x)$ its corresponding intermediate strategy. Then for every run $R^{||}$ with $\Sigma_A^{||} \models R^{||}$ there exists an extension $\hat{R}^{||} = R^{||} \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$ such that

- 1) $\forall i \in 1 \dots n : \alpha_i \in \Sigma_A^{||}(R^{||} \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{i-1}})$
- 2) $\hat{R}^{||}$ is liquidated

and there exists \hat{R}^x such that $\sum_{A}^{x} \vdash \hat{R}^x$, \hat{R}^x is liquidated and

$$\forall \mathbb{B} \in \mathcal{B} : payout_A^{\mathbb{B}}(R^{||}) - inputs_A^{\mathbb{B}}(R^{||}) \geq payout_A^{\mathbb{B}}(\hat{R^x}) - inputs_A^{\mathbb{B}}(\hat{R^x})$$

Proof. From Lemma 26, we know that there exists an extension $\hat{R}^{||} = R^{||} \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$ such that

- 1) $\forall i \in 1 \dots n : \alpha_i \in \Sigma_{\underline{A}}^{||}(R^{||} \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{i-1}})$
- 2) $\hat{R}^{||}$ is liquidated

From soundness, we can conclude that there exists also a run \hat{R}^x such that $\Sigma_A^x \vdash \hat{R}^x$ and $\hat{R}^x \sim_A \hat{R}^{|\cdot|}$. Consequently, we can use Lemma 29 to conclude that

$$\forall \mathbb{B} \in \mathcal{B} : \mathsf{payout}^{\mathbb{B}}_{A}(\hat{R^{||}}) - \mathsf{inputs}^{\mathbb{B}}_{A}(\hat{R^{||}}) \geq \mathsf{payout}^{\mathbb{B}}_{A}(\hat{R^{x}}) + \mathsf{contractFunds}^{\mathbb{B}}(\hat{R^{x}}) - \mathsf{inputs}^{\mathbb{B}}_{A}(\hat{R^{x}})$$

From Lemma 23, we know that there exists an extension $\hat{R}^x = \hat{R}^x \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$ of \hat{R}^x such that

- 1) $\forall i \in 1 \dots n : \alpha_i \in \Sigma_{\mathbf{A}}^x(\hat{R}^x \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{i-1}})$
- 2) $\not\exists \langle C, \mathbf{B} \rangle_{\kappa} \in \Gamma^{x}(\dot{R}^{x})$

So consequently,

$$\mathsf{contractFunds}^{\mathbb{B}}(\dot{R^x}) = 0$$

The claim hence immediately follows from money preservation.

Definition 36 (Liquidated Low-Level Runs). Let K be a set of contracts and R a low-level BitML run. Then R is considered liquidated w.r.t. K if for all $(C, \pi)^{\mathbb{B}}_{\kappa} \in \Gamma^{||}(R^{||})$ with $\kappa \in K$, it holds that $\exists B$. C = withdraw B.

Definition 37 (Strong Low-Level Liquidity). Let A be an honest participant with strategy Σ_A . Then Σ_A is strongly liquid if for all runs R there exists an extension $\hat{R} = R \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$ such that

- 1) $\forall i \in 1 \dots n : \alpha_i \in \Sigma_A(R \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{i-1}})$
- 2) \hat{R} is liquidated

Lemma 31 (Liquidity Connection).

$$R^{||}$$
 is liquidated $\Rightarrow f_c(R^{||})$ is liquidated

Proof. Holds by definition of translation function f_c .

Definition 38 (Low-level inputs per contract). Let R be a low-level BitML semantics run and \mathbb{B} be a blockchain. We define the low-level inputs of A in R and blockchain \mathbb{B} (written inputs $_{A:\vec{x}}^{\mathbb{B}}(R)$) as follows

$$\mathit{inputs}_{A:x}^{\mathbb{B}}(R) = \sum_{\{v: A: ! \ v^{\mathbb{B}} \ @ \ x \in G^{\mathbb{B}} \land init(\{G\}^{\mathbb{B}}\ C) \in R\}} v$$

Similarly, we define the total low-level funds per contracts:

Definition 39 (Low-level total funds per contract). Let R be a low-level BitML semantics run and \mathbb{B} be a blockchain. We define the contract funds in R (written totalFunds $_{x_0}^{\mathbb{B}}(R)$) as follows

$$\mathit{totalFunds}_{x_0}^{\mathbb{B}}(R) = \sum_{\{v: \langle C, v \rangle_x^{\mathbb{B}} \in \Gamma_R \ \land \ x \in desc(x_0, R)\}} v$$

Definition 40 (Low-level total payout per blockchain). Let R be a low-level BitML semantics run and \mathbb{B} be a blockchain. We define the low-level payout of A in R and blockchain \mathbb{B} (written $payout_A^{\mathbb{B}}(R)$) as follows

$$\textit{payout}_{A}^{\mathbb{B}}(R) = \sum_{\{v: \langle \textit{withdraw } \textit{A}, v \rangle_{x}^{\mathbb{B}} \in \Gamma_{R}\}} v + \sum_{\{v: \langle \textit{A}, v \rangle_{x}^{\mathbb{B}} \in \Gamma_{R} \ \land \ \neg initial(x)\}} v$$

Lemma 32 (Low-Level Security). Let A be an honest participant and K be a set of contract identifiers. Let Σ_A^x be an eager and deterministic intermediate semantics strategy of A bounded by K and $\Sigma_A^{||} = S(\Sigma_A^x)$ its corresponding intermediate level strategy and $\Sigma_A = S^{||}(\Sigma_A^{||})$ its corresponding low level strategy. Then for every run R with $\Sigma_A \models R$ there exists an extension $\hat{R} = R \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$ such that

- 1) $\forall i \in 1 \dots n : \alpha_i \in \Sigma_A(R \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{i-1}})$
- 2) \hat{R} is liquidated

and there exists $\hat{R}^{||}$ such that $\Sigma_A^{||} \vdash \hat{R}^{||}$, $\hat{R}^{||}$ is liquidated and

$$\forall \mathbb{B} \in \mathcal{B} : \mathsf{payout}_{A}^{\mathbb{B}}(\hat{R}) - \mathsf{inputs}_{A}^{\mathbb{B}}(\hat{R}) \geq \mathsf{payout}_{A}^{\mathbb{B}}(\hat{R}^{||}) - \mathsf{inputs}_{A}^{\mathbb{B}}(\hat{R}^{||})$$

Proof. By low-level soundness we know that there exists an intermediate level run $R^{||}$ such that $\Sigma_A^{||} \models R^{||}$ and $R^{||} \approx R$. From Lemma 26, we know that there exists an extension $\hat{R}^{||} = R^{||} \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_n}$ such that

- 1) $\forall i \in 1 \dots n : \beta_i \in \Sigma_{\mathbf{A}}^{||}(R^{||} \xrightarrow{\beta_1} \dots \xrightarrow{\beta_{i-1}})$
- 2) $\hat{R}^{||}$ is liquidated

From translation correctness, we know that the extension trace

$$R^{||} \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_n} \hat{R^{||}}$$

where $\hat{R^{||}}$ is liquidated can mirrored on $f_c(R^{||})$:

$$f_c(R^{||}) \xrightarrow{f_a(\alpha_1)} \cdots \xrightarrow{f_a(\alpha_n)} f_c(\hat{R}^{||})$$

We will show that $\alpha_1 \dots \alpha_n = f_a(\alpha_1) \dots f_a(\alpha_n)$.

We know from liquidity connection (Lemma 31) that $f_c(\hat{R}^{||})$ is liquidated as well. In particular, all contract in $f_c(\hat{R}^{||})$ have the form $\langle withdraw A, v \rangle_x^{\mathbb{B}}$. The low-level soundness active-contract-invariant states that all contract present in $\Gamma(R)$ were also present in $f_c(R^{||})$:

$$\forall \langle C, v \rangle_x^{\mathbb{B}} \in \Gamma(R) \Rightarrow \langle C, v \rangle_x^{\mathbb{B}} \in f_c(\Gamma^{||}(R^{||}))$$

From the synchronous actions lemma (Lemma 16), we can conclude that all present contract in $\hat{R}^{||}$ are liquidated as well as the non-liquidated contract were removed with the same action in $f_c(\Gamma^{||}(R_i^{||}))$ and R_i .

The intermediate to low-level soundness invariants are guaranteeing that all deposits, withdraw contracts and inputs in R are part of $f_{c,\Gamma^{||}}(R^{||})$ which can be mapped back to $R^{||}$. Therefore, we know that for R and $R^{||}$ it holds that

$$\forall \mathbb{B} \in \mathcal{B}: \mathsf{payout}^{\mathbb{B}}_{A}(R) = \mathsf{payout}^{\mathbb{B}}_{A}(f_{c}(R^{||})) = \mathsf{payout}^{\mathbb{B}}_{A}(R^{||}) \ \land \ \mathsf{inputs}^{\mathbb{B}}_{A}(R) = \mathsf{inputs}^{\mathbb{B}}_{A}(f_{c}(R^{||})) = \mathsf{inputs}^{\mathbb{B}}_{A}(R^{||})$$

From synchronous actions, we follow that balance is persevered by every action.

APPENDIX M AUXILIARY FUNCTIONS

For any contract κ in an intermediate or $BitML^x$ run:

- $desc(\kappa)$ is the set of contracts $\kappa^{\downarrow} = \kappa | \dots$
- $ldesc(\kappa)$ is the set of contracts $\kappa^{\downarrow} = \kappa |L| \dots$
- $ldesc(\kappa)$ is the set of contracts $\kappa^{\downarrow} = \kappa |R| \dots$

The following functions make it more easy to analyze the contents of contract preconditions. Suppose a contract advertisement $\{G\}C$ such that we can write G by extension as:

$$G = \prod_{i=1}^{n} \prod_{j=1}^{k} \left(A_i : ! v_{i,j} \mathbb{B}_j @ x_{i,j} \right)$$

$$\mid \prod_{i=1}^{n} \prod_{j=1}^{m} \left(A_i : \text{secret } s_{i,j} \right)$$

$$\mid t_0 \delta @ \kappa$$

Then, we can define:

$$\begin{aligned} chains(G) &= [\mathbb{B}_1, \dots, \mathbb{B}_k] \\ users(G) &= \{A_i, \forall i = 1, \dots, n\} \\ deposits(G) &= \prod_{i=1}^n \prod_{j=1}^k \left(A_i : ! \, v_{i,j} \mathbb{B}_j @ x_{i,j} \right) \\ \forall A_i. \ deposits_{A_i}(G) &= \prod_{j=1}^k \left(A : ! \, v_{i,j} \mathbb{B}_j @ x_{i,j} \right) \\ secrets(G) &= \{s_i, \forall i = 1, \dots, m\} \\ \forall A_i. \ secrets_{A_i}(G) &= \{s_i, \forall i = 1, \dots, m : \ A_i = A\} \\ \forall \mathbb{B}_j. \ balance_{\mathbb{B}_j}(G) &= \sum_{i=1}^n v_i \mathbb{B}_j \\ balance(G) &= [\sum_{i=1}^n v_{i,1} \mathbb{B}_1, \dots, \sum_{i=1}^n v_{i,k} \mathbb{B}_k] \end{aligned}$$

The following functions give us information about the history of the moves in an intermediate run $R^{||}$

$$\begin{split} \mathcal{U}_{\kappa}(R^{||}) &= \{ \textbf{A} \in \pi.participants : \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in R^{||} \} \\ \mathcal{C}_{\kappa}^{\textbf{A}}(R^{||}) &= \{ \mathbb{B} \in \mathcal{B} : \langle C, \pi \rangle_{\kappa}^{\mathbb{B}} \in R^{||} \land \pi.status = \textbf{CompensatedFrom}(B) \land \textbf{A} \in \mathcal{U}_{\kappa}(R^{||}) \backslash \{ \textbf{B} \} \} \\ \mathcal{C}_{\kappa}^{\textbf{A}}(R^{||}) &= \bigcup_{\kappa^{\uparrow} \in anc(\kappa)} \left(\mathcal{C}_{\kappa^{\uparrow}}^{\textbf{A}}(R^{||}) \cup \mathcal{C} f_{R^{||}}^{\kappa^{\uparrow}}() \right) \\ \mathcal{C}^{\textbf{A}}(R^{||}) &= \{ \kappa \in R^{||} : \mathcal{C}_{\kappa}^{\textbf{*}}(R^{||}) = \mathcal{B} \} \\ \mathcal{A}_{\kappa}(R^{||}) &= \{ \textbf{A} : \mathcal{A}_{\kappa}^{\textbf{A}}(R^{||}) \neq \emptyset \} \\ \mathcal{A}_{\kappa}(R^{x}) &= \{ \textbf{A} : (A : \kappa) \in R^{x} \} \\ \mathbb{A}\mathbb{U}_{\kappa}^{\textbf{A}}(R^{||}) &= \{ \textbf{B} \in \mathcal{A}_{\kappa}(R^{||}) : \mathcal{A}_{\kappa}^{\textbf{B}}(R^{||}) \cup \mathcal{C}_{\kappa}^{\textbf{*}}(R^{||}) = \mathcal{B} \} \\ \mathcal{R}_{\textbf{A}}^{ss}(R^{||}) &= \{ \kappa \in R^{||} : A : s_{\kappa}^{\textbf{A}} \in \Gamma_{R^{||}} \} \\ \mathcal{R}_{\textbf{A}}^{is}(R^{||}) &= \{ \kappa_{0} \in R^{||} : A : is_{\kappa_{0}}^{\textbf{A}} \in \Gamma_{R^{||}} \} \end{split}$$