VERIFICATION OF RECURSIVE BITCOIN CONTRACTS

MASSIMO BARTOLETTI, STEFANO LANDE, MAURIZIO MURGIA, AND ROBERTO ZUNINO

University of Cagliari, Italy
e-mail address: bart@unica.it

University of Cagliari, Italy
e-mail address: lande@unica.it

University of Trento, Italy
e-mail address: maurizio.murgia@unitn.it

University of Trento, Italy
e-mail address: roberto.zunino@unitn.it

ABSTRACT. Smart contracts — computer protocols that regulate the exchange of crypto-assets in trustless environments — have become popular with the spread of blockchain technologies. A landmark security property of smart contracts is *liquidity*: in a non-liquid contract, it may happen that some assets remain frozen, i.e. not redeemable by anyone. The relevance of this issue is witnessed by recent liquidity attacks to Ethereum, which have frozen hundreds of USD millions. We address the problem of verifying liquidity on BitML, a DSL for smart contracts with a secure compiler to Bitcoin, featuring primitives for currency transfers, contract renegotiation and consensual recursion. Our main result is a verification technique for liquidity. We first transform the infinite-state semantics of BitML into a finite-state one, which focusses on the behaviour of a chosen set of contracts, abstracting from the moves of the context. With respect to the chosen contracts, this abstraction is sound, i.e. if the abstracted contract is liquid, then also the concrete one is such. We then verify liquidity by model-checking the finite-state abstraction. We implement a toolchain that automatically verifies liquidity of BitML contracts and compiles them to Bitcoin, and we assess it through a benchmark of representative contracts.

1. INTRODUCTION

Smart contracts — computer protocols that regulate the exchange of assets in trustless environments — have become popular with the growth of interest in blockchain technologies. Mainstream blockchain platforms like Ethereum, Tezos and Cardano, feature expressive high-level languages for programming smart contracts. This flexibility has a drawback in that it may open the door to attacks that steal or tamper with the assets controlled by vulnerable contracts [ABC17, LCO+16].

An alternative approach is to sacrifice the expressiveness of smart contracts to reduce the attack surface. This approach was pursued first by Bitcoin, where transactions can specify simple conditions on how to redeem them, using a limited set of logic, arithmetic, and

cryptographic operators. Despite the limited expressiveness of these conditions, it is possible to encode a variety of smart contracts, e.g. gambling games, escrow services, crowdfunding systems, by suitably chaining transactions [ADMM14a, ADMM14c, ADMM16, ABC+18, BDM16, BZ17, BK14, KB14, KB16, KMB15, KVV16, MB17]. The common trait of these works is that they render contracts as cryptographic protocols, where participants can exchange/sign messages, read the blockchain, and append transactions. Verifying the correctness of these protocols is hard, since it requires to reason in a computational model, where participants can manipulate arbitrary bitstrings, only being constrained to use PPTIME algorithms.

Departing from this approach, BitML [BZ18] allows to write Bitcoin contracts in a high-level, process-algebraic language. BitML features a compiler that translates contracts into sets of standard Bitcoin transactions. The compiler enjoys a computational soundness property, which guarantees that the execution of the compiled contract is coherent with the semantics of the source BitML specification, even in the presence of adversaries.

In this paper we address the problem of verifying BitML contracts, in an extension of BitML with renegotiation and recursion [BMZ20]. In particular, we focus on a landmark property of smart contracts, called *liquidity*, which ensures that funds cannot remain frozen within a contract. Before discussing our main contributions, we overview below BitML and our analysis technique.

1.1. **BitML overview.** In BitML, any participant can broadcast a *contract advertisement* $\{G\}C$, where C is the contract, specifying the rules to transfer bitcoins ($\mathbf{B}$), while G is a set of preconditions to its stipulation. Preconditions may require participants to deposit some $\mathbf{B}$ in the contract, or to commit to some secret. Once $\{G\}C$ has been advertised, each participant can choose whether to accept it, or not. When all the preconditions G have been satisfied, and all the involved participants have accepted, the contract C becomes *stipulated*. Stipulated contracts have a balance, initially set to the sum of the deposits required by its preconditions. This balance is updated when participants execute the contract, e.g. by depositing/withdrawing funds to/from the contract.

A contract C is a *choice* among zero or more branches. Each branch is a *guarded contract*, consisting of one action, and zero or more continuations. The guarded contract `withdraw` A transfers the whole balance to A, and then terminates. The guarded contract `split` $\|_{i=1}^{n} w_i \to C_i$ decomposes the contract into n parallel components C_i, distributing the balance according to the weights w_i. The guarded contract `reveal` $\vec{a}$ `if` p checks that all the secrets $\vec{a}$ have been revealed and satisfy the predicate p. When enabled, the above-mentioned actions can be fired by anyone, at anytime. To restrict *who* can execute actions and *when*, one can use the decoration $A : D$, which requires the authorization of A, and the decoration `after` $t : D$, which requires to wait until time t. Finally, the guarded contract $*: $`rngt` $X\langle \vec{\mathcal{E}} \rangle$ renegotiates the contract, passing the control to the contract C when $X(\vec{\beta}) = \{G\}C$, and instantiating the formal parameters $\vec{\beta}$ to the actual parameters $\vec{\mathcal{E}}$. Similarly to stipulation, renegotiation requires that the precondition G is satisfied, and that all the participants accept. Note that C could refer to X, so enabling recursion.

We exemplify BitML by specifying the *timed commitment* contract, a basic protocol to construct more complex contracts, like e.g. lotteries and other games [ADMM14c]. Assume that a participant A wants to choose a secret, and promises to reveal it before some time t. The contract must ensure that if A does not reveal the secret in time, then she will pay a

penalty of 1Ƀ to B (e.g., the opponent in a game). In BitML, this is modelled as follows:

$$\{A\!:\!1\,@\,x \mid A\!:\!\texttt{secret}\,a\}\;(\texttt{reveal}\,a.\texttt{withdraw}\;A \;\;+\;\; \texttt{after}\,t\!:\!\texttt{withdraw}\;B)$$

The precondition requires A to pay upfront 1Ƀ, deposited in a transaction x, and to commit to a secret a. The contract is a choice between two branches. Only A can choose the first branch, by performing $\texttt{reveal}\,a$. Subsequently, anyone can transfer 1Ƀ to A by performing $\texttt{withdraw}\;A$. Only after time t, if the $\texttt{reveal}$ has not been fired, any participant can fire $\texttt{withdraw}\;B$ in the second branch, transferring 1Ƀ to B. So, before time t, A has the option to reveal a (avoiding the penalty), or to keep it secret (paying the penalty). If no branch is taken by time t, the first one who fires its $\texttt{withdraw}$ gets 1Ƀ.

1.2. **Liquidity.** The *liquidity* property requires that the contract balance is always eventually transferred to some participant. In a non-liquid contract, funds can be frozen forever, unavailable to anyone, hence effectively destroyed. A simple form of liquidity could just require that participant can always cooperate to unfreeze funds. However, this notion would contrast with the setting of smart contracts, where participants are mutually untrusted, and may refuse to cooperate. For instance, consider a contract where A and B contribute 1Ƀ each for a donation of 2Ƀ to either C or D (we omit the preconditions for brevity):

$$A\!:\!B\!:\!\texttt{withdraw}\;C \;\;+\;\; A\!:\!B\!:\!\texttt{withdraw}\;D$$

In order to unlock the funds, A and B must agree on the recipient of the donation, by granting their authorization on the same branch. This contract would be liquid only by assuming the cooperation between A and B: indeed, A alone cannot guarantee that the 2Ƀ will eventually be donated, as B can choose a different recipient, or even refuse to give any authorization. Consequently, unless A trusts B, it makes sense to consider this contract as non-liquid, from the point of view of A (and for similar reasons, also from that of B).

Consider now the timed commitment contract discussed before:

$$\texttt{reveal}\,a.\texttt{withdraw}\;A \;\;+\;\; \texttt{after}\,t\!:\!\texttt{withdraw}\;B$$

This contract is liquid from A's point of view, *even* if B is dishonest: indeed, A can reveal the secret and then redeem the funds from the contract. The timed commitment is also liquid from B's point of view: if A does not reveal the secret (making the first branch stuck), the funds in the contract can be redeemed through the second branch, after time t.

In a *mutual* timed commitment, where A and B have to exchange their secrets or pay a 1Ƀ penalty, achieving liquidity is a bit more challenging. We first consider a wrong attempt:

$$\texttt{reveal}\,a.\,\texttt{reveal}\,b.\,\texttt{split}\,(1Ƀ \to \texttt{withdraw}\;A \mid 1Ƀ \to \texttt{withdraw}\;B)$$
$$+\,\texttt{after}\,t\!:\!\texttt{withdraw}\;B$$

This contract is liquid for B, but not for A. Indeed, if A performs the $\texttt{reveal}\,a$ action, B could refuse to reveal b, making the contract stuck. Instead, B can wait time t and then fire $\texttt{withdraw}\;B$; if, in the meanwhile, A has fired $\texttt{reveal}\,a$, B can reveal his secret and fire $\texttt{reveal}\,b$, and then liquidate the contract.

To make the contract liquid for both participants, we amend it as follows:

$$\texttt{reveal}\,a.\,\big(\texttt{reveal}\,b.\,\texttt{split}\,(1Ƀ \to \texttt{withdraw}\;A \mid 1Ƀ \to \texttt{withdraw}\;B)$$
$$+\,\texttt{after}\,t+1\!:\!\texttt{withdraw}\;A\big)$$
$$+\,\texttt{after}\,t\!:\!\texttt{withdraw}\;B$$

Now, if A has fired `reveal` a but B refuses to reveal b, after time $t+1$ she can liquidate the contract by performing the `withdraw` A.

As a more involved example, consider a recursive variant of the timed commitment:

$$\mathtt{X}(n) = \{\mathsf{A}\colon \mathtt{secret}\, a \mid \mathsf{A}\colon 1\, @\, d\} C$$

$$C = *\colon \mathtt{rngt}\ \mathtt{X}\langle n+1\rangle\ +\ \mathtt{reveal}\, a.\mathtt{withdraw}\ \mathsf{A}\ +\ \mathtt{after}\,(t+n)\colon \mathtt{withdraw}\ \mathsf{B}$$

This contract is a toy example of a recursive contract, where $\mathtt{X}\langle n\rangle$ can be renegotiated, transferring its balance to $\mathtt{X}\langle n+1\rangle$. When this happens, A must commit to a new secret a, and provide an additional deposit d of 1Ḃ. Beyond renegotiation, $\mathtt{X}\langle n\rangle$ allows A to reveal her secret and withdraw all the bitcoins deposited in the contract so far. If she does not reveal, B can fire the last branch after time $t+n$, transferring the whole balance to himself. This contract is liquid for both A and B. In every reachable state, a participant can stop renegotiating the current contract $\mathtt{X}\langle n\rangle$. Then, anyone can liquidate the contract by either waiting until time $t+n$ and then performing the `withdraw` B, and firing the `withdraw` A when this action is enabled.

The examples above, albeit elementary, show that detecting if a contract is liquid is not straightforward, in general. Automatic techniques for the verification can be useful tools for the developers of smart contracts.

1.3. **Verifying liquidity.** One of the main contributions of this paper is a verification technique for the liquidity of BitML contracts. Our technique is based on a more general result, i.e. a correspondence between the concrete semantics of BitML and a new abstract semantics, which is finite-state and correctly approximates the concrete semantics. To obtain a finite-state abstraction, we need to cope with several sources of infiniteness of the concrete semantics: the unbounded passing of time, the stipulation and renegotiation of contracts, and the operations on bitcoin deposits. When studying the liquidity of a set of contracts X from the point of view of a participant A, we abstract away all this, by just recording the actions which can be performed on the descendants of X, distinguishing between the actions doable by A alone from those which require the cooperation of other participants. This abstraction produces a finite-state LTS, which we model-check for liquidity.

1.4. **Contributions.** We summarise our main contributions as follows:
- We introduce an extension of BitML featuring the renegotiation primitive $*\colon \mathtt{rngt}\ \mathtt{X}$. Compared to the version in [BZ18], the current language is more expressive: besides allowing participants to provide new deposits and secrets at run-time, it also allows for *unbounded* recursion, still admitting compilation to Bitcoin.
- We formalize a notion of liquidity (Definition 3.3). With respect to a participant A, a contract is liquid when A alone can ensure that funds do not remain frozen within the contract, even in the presence of adversaries.
- We introduce an abstraction of the semantics of BitML which is finite-state, and sound with respect to the concrete (infinite-state) semantics. Building upon this abstraction, we devise a sound verification technique for liquidity in BitML (Theorem 5.7).
- We develop a toolchain for writing and verifying the liquidity of BitML contracts, and for deploying them on Bitcoin. The toolchain is based on a BitML embedding in Racket [Fla12], which allows for programming BitML contracts within the DrRacket IDE. The toolchain also implements a compiler from BitML contracts to standard Bitcoin transactions.

- We implement a collection of BitML contracts, which we use as a benchmark to evaluate our toolchain. This collection contains a variety of complex contracts, including financial services, auctions, lotteries, and other gambling games.
- We discuss alternative renegotiation primitives, which allow participants to choose some parameters (e.g. the amounts to be deposited) at renegotiation time, to change the set of participants involved in the renegotiated contract, and to renegotiate contracts without the consent of all participants.
- We discuss alternative notions of liquidity, e.g. taking into account participants' strategies.

1.5. **Comparison with previous work.** This paper borrows and extends the contributions of some past papers of ours. BitML was originally introduced in [BZ18], in a version without renegotiation and recursion. A main limitation of this version was that the participants could not renegotiate the terms of a stipulated contract: this prevented from expressing common financial contracts, where funds have to be added by participants at run-time. Renegotiation and recursion were added in [BMZ20], where we showed that, despite the increased expressiveness, it was still possible to execute BitML on standard Bitcoin, preserving the security guarantees of BitML. These papers did not deal with verification of contracts, and with liquidity. This notion was introduced in [BZ19], in the original version of BitML, i.e. without renegotiation and recursion. The BitML toolchain was first presented in [ABL$^+$19], supporting the compilation and verification of contracts in the original version of BitML.

The current paper is the first one which studies liquidity in the full BitML. Recursion adds significant complexity to verification, as it makes the calculus Turing-complete. Because of this, the abstraction in [BZ19] is no longer usable, so the current paper devises an alternative verification technique. The current paper also improves the BitML toolchain, extending the compiler and the liquidity verifier to contracts with renegotiation and recursion.

2. BitML with renegotiation and recursion

We assume a set of *participants*, ranged over by $\mathsf{A}, \mathsf{B}, \ldots$, a set of *deposit names* $x, y, \ldots$, a set of *deposit variables* $d, e, \ldots$, and a set of *secret names* $a, b, \ldots$. We use $\chi, \chi', \ldots$ to range over deposits (both names and variables), and v, v', w to range over non-negative rational values. We denote with Hon the set of the *honest* participants. We denote with $\mathcal{S}_\mathsf{A}$ the set of secret names usable by A, requiring that $\mathcal{S}_\mathsf{A} \cap \mathcal{S}_\mathsf{B} = \emptyset$ if $\mathsf{A} \neq \mathsf{B}$.

Definition 2.1 (Contract precondition). Contract preconditions have the following syntax (the deposits χ in a contract precondition G must be distinct):

$$
\begin{array}{llll}
G ::= & \mathsf{A}\!:\! v \,\texttt{@}\, \chi & \text{deposit of } v\text{Ƀ put by } \mathsf{A} \\
& |\ \mathsf{A}\!:\!\texttt{secret}\, a & \text{secret committed by } \mathsf{A}\ (a \in \mathcal{S}_\mathsf{A}) \\
& |\ G \,|\, G & \text{composition} & \diamond
\end{array}
$$

The precondition $\mathsf{A}\!:\! v \,\texttt{@}\, \chi$ requires A to own vɃ in a deposit χ, and to spend it for stipulating the contract. The precondition $\mathsf{A}\!:\!\texttt{secret}\, a$ requires A to generate a secret a, and commit to it before the contract starts. After stipulation, A can choose whether to disclose the secret a, or not.

$$\begin{array}{llll}
C ::= \sum_{i \in I} D_i & \text{contract} & p ::= \mathit{true} & \text{truth} \\
D ::= & \text{guarded contract} & \mid p \wedge p & \text{conjunction} \\
\quad \mathtt{withdraw}\, \mathsf{A} & \text{transfer the balance to } \mathsf{A} & \mid \neg p & \text{negation} \\
\mid \mathtt{reveal}\, \vec{a}\, \mathtt{if}\, p.\, C & \text{reveal secrets (if } p \text{ is true)} & \mid E = E & \text{equality} \\
\mid \mathtt{split}\, \|_i\, (w_i \to C_i) & \text{split the balance} & \mid E < E & \text{less than} \\
\mid \mathsf{A} : D & \text{wait for } \mathsf{A}\text{'s authorization} & E ::= \mathcal{E} & \text{static expression} \\
\mid \mathtt{after}\, \mathcal{E} : D & \text{wait until time } \mathcal{E} & \mid a & \text{secret} \\
\mid * : \mathtt{rngt}\, \mathtt{X}\langle \vec{\mathcal{E}} \rangle & \text{renegotiate the contract} & \mid E + E & \text{addition} \\
& & \mid E - E & \text{subtraction}
\end{array}$$

FIGURE 1. Syntax of BitML contracts.

To define contracts, we assume a finite set of *recursion variables*, ranged over by $\mathtt{X}, \mathtt{Y}, \ldots$, and a language of *static expressions* $\mathcal{E}, \mathcal{E}', \ldots$, formed by integer constants k, integer variables $\beta, \beta', \ldots$, and the usual arithmetic operators. We omit to define the syntax and semantics of static expressions, since they are standard. We assume that a closed static expression evaluates to an integer value. We use the $\vec{\cdot}$ notation for finite sequences.

Definition 2.2 (**Contract**). Contracts are terms with the syntax in Figure 1, where: (i) each recursion variable $\mathtt{X}$ has a unique defining equation $\mathtt{X}(\vec{\beta}) = \{G\}C$; (ii) renegotiations $* : \mathtt{rngt}\, \mathtt{X}\langle \vec{\mathcal{E}} \rangle$ have the correct number of arguments; (iii) the names $\vec{a}$ in $\mathtt{reveal}\, \vec{a}\, \mathtt{if}\, p$ are distinct, and they include those in p. We denote with 0 the empty sum. The order of decorations is immaterial, e.g., $\mathtt{after}\, \mathcal{E} : \mathsf{A} : \mathsf{B} : D$ is equivalent to $\mathsf{B} : \mathsf{A} : \mathtt{after}\, \mathcal{E} : D$.   ◇

A contract C is a choice among guarded contracts. A guarded contract $\mathtt{reveal}\, \vec{a}\, \mathtt{if}\, p.\, C'$ continues as C' once all the secrets $\vec{a}$ have been revealed and satisfy the predicate p. The guarded contract $\mathtt{split}\, (w_1 \to C_1 \mid \cdots \mid w_n \to C_n)$ divides the contract into n contracts C_i, splitting the balance according to the weights w_i. The action $\mathtt{withdraw}\, \mathsf{A}$ transfers the whole balance to A. To restrict *who* can execute a branch and *when*, one can use the decoration $\mathsf{A} : D$, requiring to wait for A's authorization, and the decoration $\mathtt{after}\, \mathcal{E} : D$, requiring to wait until the time specified by the static expression $\mathcal{E}$. The guarded contract $* : \mathtt{rngt}\, \mathtt{X}\langle \vec{\mathcal{E}} \rangle$ allows the participants involved in the contract to renegotiate it. Intuitively, if $\mathtt{X}(\vec{\beta}) = \{G\}C$, then the contract continues as $C\{\vec{\mathcal{E}}/\vec{\beta}\}$ if all the participants give their authorization, and satisfy the precondition G.

Definition 2.3 (**Contract advertisement**). A contract advertisement is a term $\{G\}C$ such that: (i) each secret name in C occurs in G; (ii) G requires a deposit from each A in $\{G\}C$; (iii) each $* : \mathtt{rngt}\, \mathtt{X}\langle \vec{\mathcal{E}} \rangle$ in C refers to a defining equation $\mathtt{X}(\vec{\beta}) = \{G'\}C'$ where the participants in G' are the same as those in G.   ◇

Condition (ii) is used to guarantee that the contract is stipulated only if *all* the involved participants give their consent: namely, A's consent is rendered as A's authorization to spend one of her deposits. The last condition is only used to simplify the technical development: we outline in Section 8 how to relax it, by allowing renegotiations to exclude some participants, or to include new ones, which were not among those who originally stipulated the contract.

We now define the semantics of BitML, starting from its configurations.

$\Gamma ::=$	configuration		
0	empty	$\chi ::=$	authorization to …
$\mid \{G\}^x C$	contract advertisement	$\# \rhd \{G\}^\circ C$	commit to $\{G\}^\circ C$
$\mid \langle C, v \rangle_x$	active contract storing $v\textrm{\bitcoin}$	$\mid x \rhd \{G\}^\circ C$	spend x for $\{G\}^\circ C$
$\mid \langle A, v \rangle_x$	A's deposit of $v\textrm{\bitcoin}$	$\mid x \rhd D$	take branch D
$\mid A[\chi]$	A's authorization for χ	$\mid x, y \rhd \langle A, v \rangle$	join deposits x and y
$\mid \{A : a\#N\}$	committed secret of A	$\mid x \rhd \langle A, v \rangle, \langle A, v' \rangle$	split deposit x in two
$\mid A : a\#N$	revealed secret of A	$\mid x \rhd B$	donate deposit x to B
$\mid t$	global time	$\mid \vec{x}, i \rhd y$	destroy i-th deposit in $\vec{x}$
$\mid \Gamma \mid \Gamma'$	composition		

FIGURE 2. Configurations and authorizations.

Definition 2.4 (**Configuration**). Configurations are terms with the syntax in Figure 2, where: (i) in a contract advertisement $\{G\}^x C$, the name x is optional; (ii) in a committed secret, $N \in \mathbb{N} \cup \{\bot\}$ (where $\bot$ denotes an ill-formed commitment); (iii) in a revealed secret, $N \in \mathbb{N}$; (iv) in a configuration there are no duplicate authorizations; (v) in a configuration containing $\langle \cdots \rangle_x$ and $\langle \cdots \rangle_y$, it must be $x \neq y$; (vi) there exists at most one term t. We assume that $(\mid, 0)$ is a commutative monoid, and we denote indexed parallel compositions with $\|_i$. We say that Γ is *initial* when it contains only deposits (i.e., terms $\langle A, v \rangle_x$), and that it is a *timed configuration* when it contains a term t. We denote with $\mathrm{cn}(\Gamma)$ the set of contract names x such that Γ contains $\langle C, v \rangle_x$, for some C and v. ◇

The intuition behind the various terms in configurations is the following:
- $\{G\}C$ is the advertisement of a contract C with preconditions G;
- $\{G\}^x C$ is the advertisement of a renegotiation of an existing contract x;
- $\langle C, v \rangle_x$ is a stipulated contract storing $v\textrm{\bitcoin}$, uniquely identified by the name x;
- $\langle A, v \rangle_x$ is a deposit of $v\textrm{\bitcoin}$ owned by A, and uniquely identified by the name x;
- $A[\chi]$ is A's *authorizations* to perform some action χ;
- $\{A : a\#N\}$ represents A's commitment to a secret N, identified by a;
- $A : a\#N$ represents a secret N, identified by a, and revealed by A.

Definition 2.5 (**BitML semantics**). The semantics of BitML is an LTS between timed configurations. In the rest of this section we describe the reduction rules of the LTS, which implicitly define the labels ℓ. A *concrete run* $\mathcal{R}$ is a sequence $\Gamma_0 \xrightarrow{\ell_0}_\flat \Gamma_1 \xrightarrow{\ell_1}_\flat \cdots$, where Γ_0 is timed. If $\mathcal{R}$ is finite, we write $\Gamma_\mathcal{R}$ for the untimed part of its last configuration.

Below we gently introduce the BitML semantics, first illustrating each construct through examples, and then giving the general rule. Intuitively, whenever the label of a transition has the form $A : \cdots$ it means that performing that transition needs the cooperation from A. In the examples, we will omit the labels.

Deposits. A deposit $\langle A, v \rangle_x$ can be reduced in several ways: it can be split into two smaller deposits, joined with another deposit, transferred to another participant, or destroyed. In

all cases, its owner A must authorise the action. For instance, to authorize the join of two deposits, A can perform the following step:

$$\langle \mathsf{A}, v \rangle_x \mid \langle \mathsf{A}, v' \rangle_y \ \rightarrow_\flat \ \langle \mathsf{A}, v \rangle_x \mid \langle \mathsf{A}, v' \rangle_y \mid \mathsf{A}[x, y \rhd \langle \mathsf{A}, v + v' \rangle]$$

where $x, y \rhd \langle \mathsf{A}, v + v' \rangle$ is the A's authorization to spend x. After A also provides the dual authorization to spend y, anyone can join the deposits, producing a new deposit z:

$$\langle \mathsf{A}, v \rangle_x \mid \langle \mathsf{A}, v' \rangle_y \mid \mathsf{A}[x, y \rhd \langle \mathsf{A}, v + v' \rangle] \mid \mathsf{A}[y, x \rhd \langle \mathsf{A}, v + v' \rangle] \ \rightarrow_\flat \ \langle \mathsf{A}, v + v' \rangle_z$$

The reduction rules for deposits are straightforward, and they are detailed in Appendix A.

Stipulation: advertisement. Any participant can broadcast a contract advertisement $\{G\}C$, provided that all the deposits mentioned in G exist in the current configuration, and that the names of the secrets in G are fresh. This is formalised by the following rule:

$$\frac{\forall \mathsf{A} \colon v \,@\, x \text{ in } G : \langle \mathsf{A}, v \rangle_x \text{ in } \Gamma \qquad \text{all secrets in } G \text{ fresh}}{\Gamma \xrightarrow{\ adv(\{G\}C)\ }_\flat \{G\}C \mid \Gamma} \text{[C-Adv]}$$

We exemplify this and the following rules through a running example. Let:

$$G = \mathsf{A} \colon 1 \,@\, x \mid \mathsf{B} \colon 1 \,@\, d \mid \mathsf{A} \colon \mathtt{secret}\, a$$

This precondition requires both A and B to deposit $1\overset{.}{\mathsf{B}}$, but A's deposit name x is known, while B'one is not known yet, so we refer to it through a deposit variable d. Let C be an arbitrary contract involving only A and B, and let $\Gamma_0 = \Gamma \mid \langle \mathsf{A}, 1 \rangle_x \mid \langle \mathsf{B}, 1 \rangle_y$ for some Γ. By rule [C-Adv], the configuration Γ_0 can take the transition:

$$\Gamma_0 \ \rightarrow_\flat \ \Gamma_0 \mid \{G\}C \ = \ \Gamma_1$$

Stipulation: commitment. To stipulate an advertised contract, all the participants mentioned in it must fulfill the preconditions, by making available the required deposits, and committing to the required secrets. In our example, A has one secret to commit, so she can perform the following step:

$$\Gamma_1 \ \rightarrow_\flat \ \Gamma_1 \mid \{\mathsf{A} : a \# N\} \mid \mathsf{A}[\# \rhd \{G\}C] \ = \ \Gamma_2 \tag{2.1}$$

where the term $\{\mathsf{A} : a \# N\}$ represents A's commitment to the secret N, while $\mathsf{A}[\# \rhd \{G\}C]$ represents finalising the commitment phase for A. Participant B has no secrets to commit, but he must choose one of his deposits (e.g., $\langle \mathsf{B}, 1 \rangle_y$) to fulfill the precondition $\mathsf{B} \colon 1 \,@\, d$:

$$\Gamma_2 \ \rightarrow_\flat \ \Gamma_2 \mid \mathsf{B} : d \leftarrow y \mid \mathsf{B}[\# \rhd \{G\}C] \ = \ \Gamma_3 \tag{2.2}$$

In general, these steps are formalised by the following rule:

$$\frac{\begin{array}{ll}
\Delta_s = \|_{i=1}^{k} \{\mathsf{A} : a_i \# N_i\} & \Delta_d = \|_{i=1}^{h} \mathsf{A} : d_i \leftarrow x_i \\
a_1 \cdots a_k \text{ are all the secrets of } \mathsf{A} \text{ in } G & d_1 \cdots d_h \text{ are all the deposits of } \mathsf{A} \text{ in } G \\
\forall i \in 1..k : \nexists N : \{\mathsf{A} : a_i \# N\} \text{ in } \Gamma & \forall i \in 1..h : \mathsf{A} \colon v_i \,@\, d_i \text{ in } G \\
\forall i \in 1..k : \nexists N : \mathsf{A} : a_i \# N \text{ in } \Gamma & \forall i \in 1..h : \nexists x : (\mathsf{A} : d_i \leftarrow x) \text{ in } \Gamma \\
& \forall i \in 1..h : \langle \mathsf{A}, v_i \rangle_{x_i} \text{ in } \Gamma \\
\forall i \in 1..k : N_i \in \begin{cases} \mathbb{N} & \text{if } \mathsf{A} \in \mathsf{Hon} \\ \mathbb{N} \cup \{\bot\} & \text{otherwise} \end{cases} & \forall i \neq j \in 1..h : x_i \neq x_j \\
& \forall x, v, i \in 1..h : \mathsf{A} : v \,@\, x \text{ in } G \implies x_i \neq x
\end{array}}{\{G\}^\circ C \mid \Gamma \xrightarrow{\ \mathsf{A} : \{G\}^\circ C, \Delta_s \mid \Delta_d\ }_\flat \{G\}^\circ C \mid \Gamma \mid \Delta_s \mid \Delta_d \mid \mathsf{A}[\# \rhd \{G\}^\circ C]} \text{[C-AuthCommit]}$$

The rule preconditions ensures that the final configuration fulfills the conditions required by G. We use the notation $\{G\}^\circ C$ to refer to a contract advertisement of the form $\{G\}C$ or $\{G\}^x C$. The case $\{G\}C$ corresponds to the original contract stipulation, while the other case corresponds to renegotiation. In this way, we can use the same rule in both situations.

Stipulation: authorization. Back to our example, in the configuration Γ_3 of (2.2), A and B must perform an additional sequence of steps to authorize the transfer of their deposits x, y to the contract:

$$\Gamma_3 \rightarrow_\flat \Gamma_3 \mid \mathsf{A}[x \rhd \{G\}C] \rightarrow_\flat \Gamma_3 \mid \mathsf{A}[x \rhd \{G\}C] \mid \mathsf{B}[y \rhd \{G\}C] = \Gamma_4 \qquad (2.3)$$

where the terms $\mathsf{A}[x \rhd \{G\}C]$ and $\mathsf{B}[y \rhd \{G\}C]$ represent the authorizations to spend x and y for stipulation. In general, these steps are obtained through the following rule:

$$\frac{\forall \mathsf{B} \text{ in } G : \mathsf{B}[\# \rhd \{G\}^\circ C] \text{ in } \Gamma \qquad \mathsf{A} : v \, @ \, \chi \text{ in } G \qquad \Gamma \vdash \chi = x}{\{G\}^\circ C \mid \Gamma \xrightarrow{\mathsf{A}:\{G\}^\circ C, x}_\flat \{G\}^\circ C \mid \Gamma \mid \mathsf{A}[x \rhd \{G\}^\circ C]}[\text{C-AuthInitDep}]$$

The first premise requires that all participants have finalised the commitment phase. While rule [C-AuthCommit] allows a participant to add all her commitments to the configuration in a single step, each application of rule [C-AuthInitDep] allows one to authorize the spending of a single deposit. If a deposit variable d was used in G, the relation $\Gamma \vdash d = x$ ensures that the configuration contains a binding $\mathsf{A} : d \leftarrow x$. The relation $\vdash$ is defined as follows:

$$\Gamma \vdash x = x \qquad \Gamma \mid \mathsf{A} : d \leftarrow x \vdash d = x$$

Stipulation: initialization. In the configuration Γ_4 of (2.3) all the needed authorizations have been granted, so the advertisement can be turned into an active contract. This step consumes the deposits and the authorizations, and it initializes the new contract, with a fresh name z, and with a balance corresponding to the sum of all the consumed deposits:

$$\Gamma_4 \rightarrow_\flat \Gamma \mid \{\mathsf{A} : a\#N\} \mid \langle C, 2 \rangle_z$$

In general, this step is obtained through the following rule:

$$\frac{\begin{array}{c} G = \left(\|_{i \in I} \mathsf{A}_i : v_i \, @ \, x_i \right) \mid \left(\|_{i \in J} \mathsf{B}_i : v_i' \, @ \, d_i \right) \mid \left(\|_{i \in K} \mathsf{C}_i : \mathtt{secret} \, a_i \right) \qquad z \text{ fresh} \\ \Delta = \left(\|_{i \in I} \langle \mathsf{A}_i, v_i \rangle_{x_i} \right) \mid \left(\|_{i \in J} \langle \mathsf{B}_i, v_i' \rangle_{x_i'} \right) \mid \left(\|_{i \in J} \mathsf{B}_i : d_i \leftarrow x_i' \right) \mid \\ \left(\|_{i \in I} \mathsf{A}_i[x_i \rhd \{G\}C] \right) \mid \left(\|_{i \in J} \mathsf{B}_i[x_i' \rhd \{G\}C] \right) \mid \left(\|_{\mathsf{A} \in G} \mathsf{A}[\# \rhd \{G\}C] \right) \end{array}}{\{G\}C \mid \Delta \mid \Gamma \xrightarrow{init(G,C)}_\flat \langle C, \sum_{i \in I} v_i + \sum_{i \in J} v_i' \rangle_z \mid \Gamma}[\text{C-Init}]$$

Note that the part $\{G\}C \mid \Delta$ of the configuration contains all the terms that are consumed by the step. The following rules define the behaviour of a contract after stipulation.

Withdraw. Executing `withdraw A` terminates the contract, and transfers its balance to A:

$$\langle \texttt{withdraw A}, v \rangle_x \ \rightarrow_\flat \ \langle A, v \rangle_y$$

After the contract x is terminated, a fresh deposit of $v\ddot{\text{B}}$ owned by A is created. The general rule is the following:

$$\frac{y \text{ fresh}}{\langle \texttt{withdraw A}, v \rangle_x \mid \Gamma \xrightarrow{withdraw(A,v,x)}_\flat \langle A, v \rangle_y \mid \Gamma}[\text{C-Withdraw}]$$

The case where the action `withdraw A` has an alternative branch is dealt with by the rule [C-Branch], discussed below.

Split. The `split` primitive divides the contract balance in parts, each one controlled by its own contract. For instance:

$$\langle (\texttt{split } 2 \to C_1 \mid 3 \to C_2), 5 \rangle_x \ \rightarrow_\flat \ \langle C_1, 2 \rangle_y \mid \langle C_2, 3 \rangle_z$$

After this step, the new spawned contracts C_1 and C_2 are executed concurrently. The general rule is the following:

$$\frac{w = \sum_{i=1}^k w_i \qquad v_i = (v \cdot w_i)/w \qquad y_1 \cdots y_k \text{ fresh}}{\langle \texttt{split } \|_{i=1}^k (w_i \to C_i), v \rangle_x \mid \Gamma \xrightarrow{split(x)}_\flat \left(\|_{i=1}^k \langle C_i, v_i \rangle_{y_i} \right) \mid \Gamma}[\text{C-Split}]$$

Note that the weights w_i in the `split` do not represent actual $\ddot{\text{B}}$ values, but the proportion w.r.t. the contract balance. For instance:

$$\langle (\texttt{split } 2 \to C_1 \mid 3 \to C_2), 10 \rangle_x \ \rightarrow_\flat \ \langle C_1, 4 \rangle_y \mid \langle C_2, 6 \rangle_z$$

Revealing secrets. Any participant can reveal one of her secrets, using the rule:

$$\frac{N \neq \bot}{\{A : a\#N\} \xrightarrow{A:a}_\flat A : a\#N}[\text{C-AuthRev}]$$

The premise $N \neq \bot$ is needed to avoid the case where a participant does not know the secret she has committed to. Indeed, at the level of Bitcoin, commitments are represented as cryptographic hashes of bitstrings, and revealing a secret amounts to broadcasting a preimage, i.e. a value whose hash is equal to the committed value. If a participant commits to a random value, then with overwhelming probability she will not be able to provide a preimage. The label $A : a$ represents the fact that only A, the participant who performed the commitment, can fire the transition.

Reveal. The prefix `reveal` $\vec{a}$ `if` p can be fired if all the committed secrets $\vec{a}$ have been revealed, and satisfy the guard p. For instance, if $\Gamma = A : a\#N \mid B : b\#N$:

$$\langle \texttt{reveal } ab \texttt{ if } a = b. C, v \rangle_x \mid \Gamma \ \rightarrow_\flat \ \langle C, v \rangle_y \mid \Gamma$$

The general rule is the following:

$$\frac{\vec{a} = a_1 \cdots a_n \qquad \Delta = \|_{i=1}^n B_i : a_i\#N_i \qquad [\![p]\!]_\Delta = true \qquad y \text{ fresh}}{\langle \texttt{reveal } \vec{a} \texttt{ if } p. C, v \rangle_x \mid \Gamma \mid \Delta \xrightarrow{rev(\vec{a},y)}_\flat \langle C, v \rangle_y \mid \Gamma \mid \Delta}[\text{C-Rev}]$$

where the semantics of predicates $[\![p]\!]_\Delta$ is defined by the following equations:

$$[\![true]\!]_\Delta = true \qquad [\![p_1 \wedge p_2]\!]_\Delta = [\![p_1]\!]_\Delta \text{ and } [\![p_2]\!]_\Delta \qquad [\![\neg p]\!]_\Delta = \text{not } [\![p]\!]_\Delta$$

$$[\![a]\!]_\Delta = N \text{ if } \Delta \text{ contains } \mathsf{A} : a \# N \qquad [\![E_1 \bullet E_2]\!]_\Delta = [\![E_1]\!]_\Delta \bullet [\![E_2]\!]_\Delta \quad (\bullet \in \{+, -\})$$

$$[\![N]\!]_\Delta = N \qquad [\![E_1 \circ E_2]\!]_\Delta = [\![E_1]\!]_\Delta \circ [\![E_2]\!]_\Delta \quad (\circ \in \{=, <\})$$

Authorizing branches. A branch $\mathsf{A} : D$ can be taken only provided that A has granted her authorization. This can be done through the following rule:

$$\langle \mathsf{A} : D + C, v \rangle_x \mid \Gamma \xrightarrow{\mathsf{A} : (x, \mathsf{A} : D)}_\flat \langle \mathsf{A} : D + C, v \rangle_x \mid \mathsf{A}[x \rhd \mathsf{A} : D] \mid \Gamma \quad \text{[C-AuthBranch]}$$

Reducing branches. Once all the authorizations for a branch occur in the configuration, anyone can trigger the transition, provided that the time constraints (if any) are respected. For instance, if $D = \mathsf{A} : \mathtt{after}\,1000 : \mathtt{withdraw}\,\mathsf{B}$, we have the transition:

$$\langle D + C, v \rangle_x \mid \mathsf{A}[x \rhd D] \mid 1051 \to_\flat \langle \mathsf{B}, v \rangle_y \mid 1051$$

The general rule is the following:

$$\frac{\begin{array}{ll} \langle D', v \rangle_x \mid \Gamma \xrightarrow{\ell}_\flat \Gamma' & D = \mathsf{A}_1 : \cdots : \mathsf{A}_k : \mathtt{after}\,t_1 : \cdots : \mathtt{after}\,t_m : D' \quad D' \neq \mathsf{A} : \cdots \\ x \notin \mathrm{cn}(\Gamma') & t \geq t_1, \ldots, t_m \qquad\qquad\qquad\qquad\qquad\qquad D' \neq \mathtt{after}\,t' : \cdots \end{array}}{\langle D + C, v \rangle_x \mid \parallel_{i=1}^k \mathsf{A}_i[x \rhd D] \mid \Gamma \mid t \xrightarrow{\ell}_\flat \Gamma' \mid t} \quad \text{[C-Branch]}$$

Delaying. In any configuration, we always allow time to advance:

$$\frac{\delta > 0}{\Gamma \mid t \xrightarrow{\delta}_\flat \Gamma \mid t + \delta} \quad \text{[C-Delay]}$$

Renegotiation: advertisement. Contract renegotiation is similar to stipulation, including advertisement, commitment, authorization, and contract initialization. We illustrate these phases through a running example. Consider a configuration:

$$\Gamma_0 = \langle * : \mathtt{rngt}\,\mathtt{X}\langle k \rangle + C_{alt}, v \rangle_x \mid \Gamma \qquad\qquad \text{where } \mathtt{X}\langle \beta \rangle = \{G\}C$$

where C_{alt} contains the branches alternative to $\mathtt{rngt}$, $G = \mathsf{A} : 1\,@\,d \mid \mathsf{B} : 1\,@\,e \mid \mathsf{A} : \mathtt{secret}\,a$, and C is an arbitrary contract involving only A and B, possibly containing the integer variable β in static expressions. The renegotiation is advertised as follows:

$$\Gamma_0 \to_\flat \Gamma_0 \mid \{G'\}^x C' = \Gamma_1 \qquad\qquad\qquad (2.4)$$

where $\{G'\}^x C'$ is obtained by transforming $\{G\}C$ as follows: (i) variables d, e are renamed into fresh ones d', e', and similarly the secret name a into a', (ii) the static expressions in C are evaluated, assuming $\beta = k$, and replaced with their results. The superscript x in

$\{G'\}^x C'$ is used to record that, when the renegotiation is concluded, the contract x must be closed. In general, this step is defined through the following rule:

$$
\frac{
\begin{array}{ll}
\{G\}C \equiv \mathtt{X}\langle\vec{\mathcal{E}}\rangle & \forall\{G'\}^x C' \text{ in } \Gamma : \{G'\}C' \not\equiv \mathtt{X}\langle\vec{\mathcal{E}}\rangle \\
\text{all secrets in } G \text{ fresh} & \\
\forall\mathsf{A}\colon v \,@\, d \text{ in } G : d \text{ fresh} & \forall\mathsf{A}_i\colon v_i \,@\, x_i \text{ in } G : \langle\mathsf{A}_i, v_i\rangle_{x_i} \text{ in } \Gamma
\end{array}
}{
\begin{array}{l}
\langle * : \mathtt{rngt}\ \mathtt{X}\langle\vec{\mathcal{E}}\rangle + C_{alt}, v\rangle_x \mid \Gamma \xrightarrow{\ advRngt(\{G\}^x C)\ }_\flat \\
\langle * : \mathtt{rngt}\ \mathtt{X}\langle\vec{\mathcal{E}}\rangle + C_{alt}, v\rangle_x \mid \Gamma \mid \{G\}^x C
\end{array}
} \ [\text{C-AdvRngt}]
$$

The relation $\{G\}C \equiv \mathtt{X}\langle\vec{\mathcal{E}}\rangle$ used in the rule premise holds when, for some G', C', there exists a defining equation $\mathtt{X}(\vec{\beta}) = \{G'\}C'$ such that $\{G\}C$ is the transformation of $\{G'\}C'$ obtained by instantiating the formal parameters with the actual ones, and by α-converting the secret names and deposit variables, as done e.g. in Equation (2.4). Before formalising this transformation below in Equation (2.5), we need a few auxiliary notions. We denote with $[\![\mathcal{E}]\!]$ the evaluation of a closed static expression $\mathcal{E}$ (the actual definition is standard, so we omit it). We overload $[\![-]\!]$ to contracts and contract advertisements: namely, $[\![C]\!]$ is the contract obtained by substituting all the occurring static expressions with their valuation, and $[\![\{G\}C]\!] = \{G\}[\![C]\!]$. Then, we denote with $\{G\}C \equiv_\alpha \{G'\}C'$ the α-equivalence between two contract advertisements w.r.t. secret names and deposit variables. Finally, we define:

$$\{G\}C \equiv \mathtt{X}\langle\vec{\mathcal{E}}\rangle \iff \exists G', C' : \mathtt{X}(\vec{\beta}) = \{G'\}C' \text{ and } [\![\{G'\}C'\{[\![\vec{\mathcal{E}}]\!]/\vec{\beta}\}]\!] \equiv_\alpha \{G\}C \qquad (2.5)$$

Renegotiation: commitment. In the subsequent steps participants choose the actual deposit names, and A commits to her secret. If A owns in Γ a deposit $\langle\mathsf{A}, 1\rangle_y$, she can choose $d' = y$ to satisfy the precondition G. Similarly, B can choose $e' = z$ if he owns such a deposit in Γ. These choices are performed as follows:

$$
\begin{array}{lr}
\Gamma_1 \to_\flat \Gamma_1 \mid \mathsf{A} : d' \leftarrow y \mid \{\mathsf{A} : a'\#N\} \mid \mathsf{A}[\# \rhd \{G'\}^x C'] & = \Gamma_2 \\
\Gamma_2 \to_\flat \Gamma_2 \mid \mathsf{B} : e' \leftarrow z \mid \mathsf{B}[\# \rhd \{G'\}^x C'] & = \Gamma_3
\end{array}
$$

The general case is defined by the rule [C-AuthCommit] described before.

Renegotiation: authorization. At this point, participants must authorise to spend their deposits and the balance of the contract at x. This is done through a series of steps:

$$
\begin{array}{lr}
\Gamma_3 \to_\flat \Gamma_3 \mid \mathsf{A}[y \rhd \{G'\}^x C'] & \\
\quad \to_\flat \Gamma_3 \mid \mathsf{A}[y \rhd \{G'\}^x C'] \mid \mathsf{A}[x \rhd \{G'\}^x C'] & = \Gamma_4 \\
\quad \to_\flat \Gamma_4 \mid \mathsf{B}[z \rhd \{G'\}^x C'] & \\
\quad \to_\flat \Gamma_4 \mid \mathsf{B}[z \rhd \{G'\}^x C'] \mid \mathsf{B}[x \rhd \{G'\}^x C'] & = \Gamma_5
\end{array}
$$

The general case is defined by the rule [C-AuthInitDep] described above.

Renegotiation: initialization. Finally, the renegotiated contract is stipulated. This amounts to closing the old contract x, consuming the deposits y and z, and transferring the balance v of the old contract to the new one, which is given a fresh name x':

$$\Gamma_5 \;\to_\flat\; \langle C', v+2 \rangle_{x'} \mid \Gamma'$$

where Γ' is Γ without the deposits y and z. Note that the branches in C_{alt} are discarded only in the last step above, where we complete the renegotiation. Before this step, it would have been possible to take one of the branches in C_{alt}, aborting the renegotiation.

The initialization step is defined by the following rule, which is analogous to [C-Init]:

$$
\frac{
\begin{array}{c}
\{G\}C \equiv \mathtt{X}\langle\vec{\mathcal{E}}\rangle \qquad y \text{ fresh} \\
G = \big(\,\|_{i \in I} \mathsf{A}_i\colon v_i \, \texttt{@} \, x_i \big) \mid \big(\,\|_{i \in J} \mathsf{B}_i\colon v_i' \, \texttt{@} \, d_i \big) \mid \big(\,\|_{i \in K} \mathsf{C}_i\colon\texttt{secret}\, a_i \big) \\
\Delta = \big(\,\|_{i \in I} \langle \mathsf{A}_i, v_i \rangle_{x_i} \big) \mid \big(\,\|_{i \in J} \langle \mathsf{B}_i, v_i' \rangle_{x_i'} \big) \mid \big(\,\|_{i \in J} \mathsf{B}_i : d_i \leftarrow x_i' \big) \mid \\
\big(\,\|_{i \in I} \mathsf{A}_i[x_i \rhd \{G\}^x C] \big) \mid \big(\,\|_{i \in J} \mathsf{B}_i[x_i' \rhd \{G\}^x C] \big) \mid \\
\big(\,\|_{\mathsf{A} \in G} \mathsf{A}[\# \rhd \{G\}^x C] \mid \mathsf{A}[x \rhd \{G\}^x C] \big)
\end{array}
}{
\begin{array}{c}
\langle * : \texttt{rngt}\, \mathtt{X}\langle\vec{\mathcal{E}}\rangle + C_{alt}, v \rangle_x \mid \{G\}^x C \mid \Gamma \mid \Delta \xrightarrow{\;rngt(x,G,C)\;}_\flat \\
\langle C, v + \sum_{i \in I} v_i + \sum_{i \in J} v_i' \rangle_y \mid \Gamma
\end{array}
} \;\text{[C-Rngt]}
$$

The main difference between [C-Init] and [C-Rngt] is that the latter transfers the balance v of the old contract to the new one.

3. Liquidity

In this section we formalise a notion of contract liquidity. Aiming at generality, we parameterise this notion over:

- an LTS $\to$, which models the contract behaviour;
- a subset $\mathcal{L}_\mathsf{A}$ of the labels of the LTS $\to$, which represents the moves that can be performed by an *honest* participant A, without requiring the cooperation of the other participants.

Once these parameters are fixed, we define when, in a configuration Γ of the LTS, a set X of contracts in Γ is liquid. Roughly, this happens when the honest participant A can always make the funds stored by the contracts X be transferred to some participant. In the meanwhile, the other participants may play against her, e.g. by not revealing some secrets, or by not granting their authorizations for some branch. Note that by suitably instantiating the parameters $\to$ and $\mathcal{L}_\mathsf{A}$, we will be able to use the same notion of liquidity both with the concrete and with the abstract BitML semantics. For instance, for the concrete semantics we choose $\to \,=\, \to_\flat$, and as $\mathcal{L}_\mathsf{A}$ all the labels except those of the form $\mathsf{B} : \ell$ with $\mathsf{B} \neq \mathsf{A}$.

We start by introducing an auxiliary partial function $orig_\Gamma(\mathcal{R}, x)$ that, given a contract name x and a run $\mathcal{R}$ starting from Γ, determines the (unique) ancestor y of x in Γ, if any. Intuitively, $orig_\Gamma(\mathcal{R}, x) = y$ means that y has evolved along the run $\mathcal{R}$, eventually leading to x, and possibly to other contracts. In BitML, this happens in one of the following cases. First, a `split` can spawn new contracts, e.g.:

$$\langle \texttt{split}\, (v_1 \to C_1 \mid v_2 \to C_2), v_1 + v_2 \rangle_x \xrightarrow{\;split(x)\;}_\flat \langle C_1, v_1 \rangle_{y_1} \mid \langle C_2, v_2 \rangle_{y_2}$$

Here, both y_1 and y_2 have x as ancestor. Second, `reveal` reduces as follows:

$$\langle \texttt{reveal}\, a.\, C, v \rangle_x \mid \cdots \xrightarrow{\;reveal(a,x)\;}_\flat \langle C, v \rangle_y \mid \cdots$$

$$
\begin{aligned}
orig_\Gamma(\Gamma, x) \;&=\; x \quad \text{if } x \in \mathrm{cn}(\Gamma) \\[2mm]
orig_\Gamma(\mathcal{R} \xrightarrow{\ell} \Gamma', x) \;&=\; \begin{cases} orig_\Gamma(\mathcal{R}, x) & \text{if } x \in \mathrm{cn}(\Gamma_\mathcal{R}) \cap \mathrm{cn}(\Gamma') \\ orig_\Gamma(\mathcal{R}, y) & \text{if } x \in \mathrm{cn}(\Gamma') \setminus \mathrm{cn}(\Gamma_\mathcal{R}) \text{ and } \{y\} = \mathrm{cn}(\Gamma_\mathcal{R}) \setminus \mathrm{cn}(\Gamma') \end{cases}
\end{aligned}
$$

FIGURE 3. Origin of a contract name within a run.

In this case, the ancestor of y is x. Third, a `rngt` evolves as follows:

$$
\langle * : \mathtt{rngt}\; \mathtt{X}\langle \vec{\mathcal{E}} \rangle, v \rangle_x \mid \cdots \xrightarrow{\;init(x,G,C)\;}_\flat \langle C, v + \cdots \rangle_y \mid \cdots
$$

Also in this case, the ancestor of y is x.

Definition 3.1 (**Origin**). Let $\mathcal{R}$ be a run starting from Γ, and let x be a contract name. We define the partial function $orig_\Gamma(\mathcal{R}, x)$ by induction on the length of $\mathcal{R}$ in Figure 3.

Note that, in Figure 3, the condition $x \in \mathrm{cn}(\Gamma') \setminus \mathrm{cn}(\Gamma_\mathcal{R})$ checks that the name x has been introduced in the last transition of the run. Dually, $\{y\} = \mathrm{cn}(\Gamma_\mathcal{R}) \setminus \mathrm{cn}(\Gamma')$ checks that y has been consumed. The use of a singleton $\{y\}$ is justified by the fact that in BitML each transition can consume at most one contract name.

Example 3.2. Let $\Gamma = \langle C_1, v \rangle_y \mid \langle \mathsf{A}, v \rangle_z$, and let $\mathcal{R}$ be the following run starting from Γ, where the contracts C_1 and C_2 are immaterial, but for the fact that they enable the displayed moves:

$$
\langle C_1, v \rangle_y \mid \langle \mathsf{A}, v \rangle_z \to_\flat \langle C_1, v \rangle_y \mid \langle \mathsf{A}, v \rangle_z \mid \{G\} C_2 \to_\flat^* \langle C_1, v \rangle_y \mid \langle C_2, v \rangle_x
$$

$$
\xrightarrow{\;split(x)\;}_\flat \langle C_1, v \rangle_y \mid \langle C_2{}', v \rangle_{x'}
$$

$$
\xrightarrow{\;split(y)\;}_\flat \langle C_1{}', v' \rangle_{y'} \mid \langle C_1{}'', v - v' \rangle_{y''} \mid \langle C_2{}', v \rangle_{x'}
$$

We have that $orig_\Gamma(\mathcal{R}, y') = orig_\Gamma(\mathcal{R}, y'') = y$, since the corresponding contracts have been obtained through a split of the ancestor y, which was in Γ. Instead, $orig_\Gamma(\mathcal{R}, x')$ is undefined, because its ancestor x is not in Γ. Further, $orig_\Gamma(\mathcal{R}, y) = y$, while $orig_\Gamma(\mathcal{R}, x)$ is undefined.

Before formalising liquidity, we give some further intuition. Assume that A is an honest participant, who cares about the liquidity of a set of contracts X in Γ. After an arbitrary sequence of transitions $\Gamma \to \cdots \to \Gamma'$, where any participant may perform actions, A wants to liquidate all the contracts in Γ' originating from X, transferring their funds to participants' deposits. We want A to be able to liquidate contracts *without* the help of the other participants.

Definition 3.3 (**Liquidity**). Let $\to$ be an LTS, let $\mathcal{L}_\mathsf{A}$ be a subset of its labels, let Γ be a configuration of the LTS, and let x be a contract name. We say that:

- x is *liquidable* in Γ if there exists a run $\mathcal{R} = \Gamma \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_n} \Gamma'$ such that:
  (1) for all $i \in 1..n$, $\ell_i \in \mathcal{L}_\mathsf{A}$;
  (2) there exists no $z \in \mathrm{cn}(\Gamma')$ such that $orig_\Gamma(\mathcal{R}, z) = x$.
- x is *liquid* in Γ if, for all runs $\mathcal{R} = \Gamma \to \cdots \to \Gamma'$, all the contract names y with $orig_\Gamma(\mathcal{R}, y) = x$ are liquidable in Γ'.

We extend this to sets X of contracts names: X is liquid in Γ iff all $x \in X$ are liquid in Γ.

Intuitively, a contract x is liquidable by A when A can perform a sequence of transitions which eventually lead to a configuration containing no contract names originated from x. Consequently, all the funds in the contract x have been transferred to participants' deposits. When liquidating x, A's moves can not reveal secrets of other participants, or generate authorizations for them: A must be able to unfreeze the funds on her own, performing actions $\ell_i \in \mathcal{L}_A$. Note that if $x \notin \mathrm{cn}(\Gamma)$, then x is trivially liquidable in Γ.

The notion of liquidity is based upon that of liquidability. A contract x is liquid in a configuration Γ when, after an arbitrary sequence of moves performed by *any* participant, the contract names originated by x are liquidable by A.

We remark that, although Definition 3.3 will be instantiated with the semantics of BitML, the basic concepts it relies upon (runs, origin of contracts, moves of a participant) are quite general. Hence, our notion of liquidity, as well as the variants proposed in Section 7, can be applied to other languages for smart contracts, using their transition semantics.

Example 3.4. Recall the timed commitment contract TC from Section 1:

$$\{A\!:\!1\,@\,z \mid A\!:\!\mathtt{secret}\,a\}\,(\mathtt{reveal}\,a.\,\mathtt{withdraw}\,A \;\; + \;\; \mathtt{after}\,t\!:\!\mathtt{withdraw}\,B)$$

Let $\Gamma = \langle TC, 1 \rangle_x \mid \{A : a\#N\}$ be a configuration where the contract has been stipulated. We show that x is liquid in Γ with respect to any participant. In the configurations reachable from Γ, the contract TC has not progressed, or it has reduced to $\mathtt{withdraw}\,A$. In the first case, x is liquidable by anyone, by firing $\mathtt{withdraw}\,B$ after time t (alternatively, A may reveal the secret and then fire $\mathtt{withdraw}\,A$). Instead, if the contract has reduced to $\mathtt{withdraw}\,A$, anyone can liquidate it. Since all the descendants of x are liquidable, x is liquid.

Example 3.5. Let $G = A\!:\!1\,@\,y \mid B\!:\!1\,@\,z \mid A\!:\!\mathtt{secret}\,a$. Consider the following contracts, where p is an arbitrary predicate on a:

$$C_1 = \mathtt{reveal}\,a\,\mathtt{if}\,p.\,\mathtt{withdraw}\,A + \mathtt{reveal}\,a\,\mathtt{if}\,\neg p.\,\mathtt{withdraw}\,B$$

$$C_2 = \mathtt{split}\,\big(1 \to \mathtt{reveal}\,a\,\mathtt{if}\,p.\,\mathtt{withdraw}\,A \;\mid\; 1 \to \mathtt{reveal}\,a\,\mathtt{if}\,\neg p.\,\mathtt{withdraw}\,B\big)$$

$$C_3 = \mathtt{withdraw}\,A + B : \mathtt{split}\,\big(1 \to B : \mathtt{withdraw}\,A \mid 1 \to \mathtt{withdraw}\,B\big)$$

For $i \in \{1, 2, 3\}$, let $\Gamma_i = \langle C_i, 2 \rangle_x \mid \{A : a\#N\}$, with $N \neq \bot$. We have that:

- x is liquid in Γ_1 for A, but not for any other participant. C_1 has three reducts: C_1 itself, $\mathtt{withdraw}\,A$, and $\mathtt{withdraw}\,B$. The last two contracts are trivally liquidable by anyone. Instead C_1 is liquidable only by A, by revealing a and firing a $\mathtt{reveal}$ branch (since their guards are p and $\neg p$, one of them will be enabled), and finally firing the corresponding $\mathtt{withdraw}$. Instead, from B's viewpoint x is not liquid, because A could refuse to reveal.
- x is not liquid in Γ_2 for anyone. Indeed, none of the reducts of C_2 is liquidable, because one of the two $\mathtt{reveal}$ branches is stuck.
- x is liquid in Γ_3 for B, but not for A. Indeed, if B authorizes and performs the $\mathtt{split}$, then the reduct is not liquidable by A, since B could deny his second authorization.

Note that liquidability in Definition 3.3 assumes a sort of fairness: it requires that the moves $\ell_1 \ldots \ell_n$ are performed atomically, effectively forbidding the adversary to interfere. Atomicity might be realistic in some blockchains, but not in others. For instance, in Ethereum a sequence $\ell_1 \ldots \ell_n$ of method calls can be performed atomically: this requires to deploy a new contract with a suitable method which performs the calls $\ell_1 \ldots \ell_n$ in sequence, and then to invoke it. Bitcoin, instead, does not support atomic sequences of transactions: an honest participant could start to perform the sequence, but at some point in the middle

the adversary can interfere. Repeated interference could lead to an infinite run, where each attempt by the honest participant is hindered by the adversary. However, this is not an issue in BitML, for the following reason. Liquidity requires that any descendent of the contract must be liquidable by A *alone*. This means that A can do that by performing a sequence of moves which do not include any renegotiation, since renegotiations can be finalized only with the cooperation of all the other participants. Without renegotiation, the contract eventually terminates, so an adversary can interfere at most a finite number of times. After the last interference, A has still a way of terminating the contract, and she can do this alone, satisfying item (2) of Definition 3.3.

4. Case studies

In this section we illustrate BitML and liquidity through a few example contracts. The automatic verification of these contracts will be discussed later on in Section 6.

A fair lottery. Consider a lottery between two players. The preconditions require A and B to commit to one secret each (a and b, respectively), and to put a deposit of 3$\dot{B}$ each (1$\dot{B}$ as a bet, and 2$\dot{B}$ as a penalty for dishonest behaviour):

$$Lottery(Win) = \texttt{split}\big($$
$$2 \to (\texttt{reveal } b \texttt{ if } 0 \leq b \leq 1.\ \texttt{withdraw B}) + (\texttt{after } t : \texttt{withdraw A})$$
$$\mid 2 \to (\texttt{reveal } a.\ \texttt{withdraw A}) + (\texttt{after } t : \texttt{withdraw B})$$
$$\mid 2 \to Win\big)$$
$$Win = \texttt{reveal } a\, b \texttt{ if } a = b.\ \texttt{withdraw A} + \texttt{reveal } a\, b \texttt{ if } a \neq b.\ \texttt{withdraw B}$$

The contract splits the balance in three parts, of 2$\dot{B}$ each. The first part allows B to reveal b and then redeem 2$\dot{B}$; otherwise, after the deadline A can redeem B's penalty (as in the timed commitment). Similarly, the second part allows A to redeem 2$\dot{B}$ by revealing a. To determine the winner we compare the secrets, in the subcontract Win: A wins if the secrets are the same, otherwise B wins. This lottery is *fair*, since: (i) if both players are honest, then they will reveal their secrets within the deadline (redeeming 2$\dot{B}$ each), and then they will have a 1/2 probability of winning[1]; (ii) if a player is dishonest, not revealing the secret, then the other player has a positive payoff, since she can redeem 4$\dot{B}$.

Although fair, the lottery is *not* liquid, neither from A's nor from B's point of view, because if one of the two players does not reveal her secret, then the Win subcontract is not liquidable, and so the 2$\dot{B}$ stored therein are frozen. We can recover liquidity by replacing Win with the following contract, where $t' > t$:

$$Win_2 = Win + (\texttt{after } t' : \texttt{reveal } a.\ \texttt{withdraw A}) + (\texttt{after } t' : \texttt{reveal } b.\ \texttt{withdraw B})$$

Now, even if one of the two players does not reveal, the honest player fire her `reveal` at time t', liquidating the 2$\dot{B}$ stored in Win_2.

---

[1]Note that B could increase his probability to win the lottery by choosing a secret $N > 1$. However, doing so will make B lose his 2$\dot{B}$ deposit in the first part of `split`, and so B's *average* payoff would be negative.

Zero-coupon bonds. A zero-coupon bond [JES00] is a financial contract where an investor A pays 1Ƀ upfront to a bank B, and receives back 2Ƀ after a maturity date (say, year 2030). We can express this contract in BitML as follows. The contract precondition requires A to provide a deposit x of 1Ƀ, and B to provide a deposit of 2Ƀ. The contract is as follows:

$$ZCB \;=\; \texttt{split} \left(1 \to \texttt{withdraw B} \;\mid\; 2 \to \texttt{after}\,2030\,\texttt{:withdraw A}\right)$$

Upon stipulation, all the deposits required in the preconditions pass under the control of ZCB, and can no longer be spent by A and B. The contract splits these funds in two parts: 1Ƀ, that can be withdrawn by B at any moment, and 2Ƀ, that can be withdrawn by A after the maturity date.

Although ZCB correctly implements the functionality of zero-coupon bounds, it is quite impractical: for the whole period from the stipulation to the maturity date, 2Ƀ are frozen within the contract, and cannot be used by the bank in any way. Although this is a desirable feature for the investor, since it guarantees that he will receive 2Ƀ even if the bank bankrupts, it is quite undesirable for the bank. In the real world, the bank would be free to use its own funds, together with those of investors, to make further financial transactions through which to repay the investments. The risk that the bank bankrupts is mitigated by external mechanisms, like insurances or government intervention.

To overcomes this issue, we can exploit renegotiation. We first revise the precondition, which now requires only A's deposit. The revised contract is:

$$ZCB2 \;=\; \texttt{split} \left(1 \to \texttt{withdraw B} \;\mid\; 0 \to * \,\texttt{:rngt X}\right)$$
$$\texttt{X} \;=\; \{\texttt{B:}\,2\,\texttt{@}\,d\} \;\texttt{after}\,2030\,\texttt{:withdraw A}$$

As before, the bank can withdraw 1Ƀ at any moment after stipulation. In the second part of the split, the participants renegotiate the contract: if they both agree, 0Ƀ pass under the control of the contract X. The precondition of X requires the bank to provide 2Ƀ in a fresh deposit; upon renegotiation, A can withdraw 2Ƀ after the maturity date. The crucial difference with ZCB is that the deposit variable d is instantiated at *renegotiation* time, unlike x, which must be fixed at *stipulation* time.

The revised contract $ZCB2$ solves the problem of ZCB, in that it no longer freezes 2Ƀ for the whole duration of the bond: the bank could choose to renegotiate the contract, paying 2Ƀ, just before the maturity date. This flexibility comes at a cost, since A loses the guarantee to eventually receive 2Ƀ. To address this issue we need to add, as in the real world, an external mechanism. More specifically, we assume an insurance company I that, for an annual premium of pɃ paid by the bank, covers a face amount of fɃ (with $2 > f > 10p$):

$$\texttt{A:}\,1\,\texttt{@}\,x_1 \;\mid\; \texttt{B:}\,p\,\texttt{@}\,x_2 \;\mid\; \texttt{I:}\,f\,\texttt{@}\,x_3$$

We revise the bond contract as follows:

$$
\begin{aligned}
ZCB3 \;=\;& \texttt{split}\,\big(1 \to \texttt{withdraw}\ \mathsf{B} \\
& \qquad\quad |\ p \to \texttt{withdraw}\ \mathsf{I} \\
& \qquad\quad |\ f \to * : \texttt{rngt}\ \mathtt{X}\langle 1\rangle + \texttt{after}\,2021 : \texttt{withdraw}\ \mathsf{A}\big) \\
\mathtt{X}\langle n \in 1..9\rangle \;=\;& \{\mathsf{B}\!:\!p\,@\,d\} \\
& \texttt{split}\,\big(p \to \texttt{withdraw}\ \mathsf{I} \\
& \qquad\quad |\ f \to * : \texttt{rngt}\ \mathtt{X}\langle n+1\rangle + \texttt{after}\,(2021+n) : \texttt{withdraw}\ \mathsf{A}\big) \\
\mathtt{X}\langle 10\rangle \;=\;& \{\mathsf{B}\!:\!2\,@\,d\} \\
& \texttt{split}\,\big(f \to \texttt{withdraw}\ \mathsf{I} \\
& \qquad\quad |\ 2 \to \texttt{after}\,2030 : \texttt{withdraw}\ \mathsf{A}\big)
\end{aligned}
$$

The contract starts by transferring $1\dot{\mathsf{B}}$ to the bank, and the first year of the premium to the insurer. The remaining $f\dot{\mathsf{B}}$ are transferred to the renegotiated contract $\mathtt{X}\langle 1\rangle$, or, if the renegotiation is not completed by 2021, to the investor.

The contracts $\mathtt{X}\langle n\rangle$, for $n \in 1..9$, allow the insurer to receive the annual premium until 2030: if the bank does not renegotiate the contract for the following year (paying the corresponding premium), then the investor can redeem the face amount of $f\dot{\mathsf{B}}$. Finally, the contract $\mathtt{X}\langle 10\rangle$ can be triggered if the bank deposits the $2\dot{\mathsf{B}}$: when this happens, the face amount is given back to the insurer, and the investor can redeem $2\dot{\mathsf{B}}$ after the maturity date.

Compared to $ZCB2$, the contract $ZCB3$ offers more protection to the investor. To see why, we must evaluate A's payoff for all the possible behaviours of the other participants. If B and I are both honest, then A will redeem $2\dot{\mathsf{B}}$, as in the ideal contract ZCB. Instead, if either B or I do not accept to renegotiate some $\mathtt{X}\langle n\rangle$, then A can redeem $f\dot{\mathsf{B}}$ as a partial compensation (unlike in $ZCB2$, where A just loses $1\dot{\mathsf{B}}$). In the real world, A could use this compensation to cover the legal fee to sue the bank in court; also, I could e.g. increase the premium for future interactions with B. By further refining the contract, we could model these real-world mechanisms as oracles, which sanction dishonest participants according to the evidence collected in the blockchain and in messages broadcast by participants. For instance, if B and I accept the renegotiation $\mathtt{X}\langle n\rangle$ but A does not, then the oracle would be able to detect A's dishonesty by inspecting the authorizations broadcast in year $2021+n$. The sanction could consist e.g. in preventing A from buying other bonds from B.

Despite the apparent complexity, the contract is liquid for any participant: indeed, any potentially blocking renegotiation has an alternative timeout branch, which allows to liquidate the contract.

A fair recursive coin flipping game. Consider a simple game where two players repeatedly flip coins, and the one who wins two consecutive flips takes the pot. The precondition requires each player to deposit $3\dot{\mathsf{B}}$ and choose a secret:

$$\mathsf{A}\!:\!3\,@\,x \ \mid\ \mathsf{A}\!:\!\texttt{secret}\,a \ \mid\ \mathsf{B}\!:\!3\,@\,y \ \mid\ \mathsf{B}\!:\!\texttt{secret}\,b$$

The contract CFG (Figure 4) asks B to reveal his secret first: if B waits too much, A can withdraw the contract funds after time 1. Then, it is A's turn to reveal (before time 2, otherwise B can withdraw the funds). The current flip winner is A if the secrets

$$CFG = \texttt{reveal}\,b\,\texttt{if}\,0 \leq b \leq 1.\big($$
$$\texttt{reveal}\,ab\,\texttt{if}\,a = b.\,(*:\texttt{rngt}\,\mathsf{X_A}\langle 1\rangle\,+\,\texttt{after}\,3:Split_\mathsf{A})$$
$$+\,\texttt{reveal}\,ab\,\texttt{if}\,a \neq b.\,(*:\texttt{rngt}\,\mathsf{X_B}\langle 1\rangle\,+\,\texttt{after}\,3:Split_\mathsf{B})$$
$$+\,\texttt{after}\,2:\texttt{withdraw}\,\mathsf{B}\big)$$
$$+\,\texttt{after}\,1:\texttt{withdraw}\,\mathsf{A}$$

$$\mathsf{X_A}\langle n\rangle = \{\mathsf{A}:\texttt{secret}\,a \mid \mathsf{B}:\texttt{secret}\,b\}$$
$$\texttt{reveal}\,b\,\texttt{if}\,0 \leq b \leq 1.\big($$
$$\texttt{reveal}\,ab\,\texttt{if}\,a = b.\,\texttt{withdraw}\,\mathsf{A}$$
$$+\,\texttt{reveal}\,ab\,\texttt{if}\,a \neq b.\,(*:\texttt{rngt}\,\mathsf{X_B}\langle n+1\rangle\,+\,\texttt{after}\,(3n+3):Split_\mathsf{B})$$
$$+\,\texttt{after}\,(3n+2):\texttt{withdraw}\,\mathsf{B}\big)$$
$$+\,\texttt{after}\,(3n+1):\texttt{withdraw}\,\mathsf{A}$$

$$\mathsf{X_B}\langle n\rangle = \{\mathsf{A}:\texttt{secret}\,a \mid \mathsf{B}:\texttt{secret}\,b\}$$
$$\texttt{reveal}\,b\,\texttt{if}\,0 \leq b \leq 1.\big($$
$$\texttt{reveal}\,ab\,\texttt{if}\,a = b.\,(*:\texttt{rngt}\,\mathsf{X_A}\langle n+1\rangle\,+\,\texttt{after}\,(3n+3):Split_\mathsf{A})$$
$$+\,\texttt{reveal}\,ab\,\texttt{if}\,a \neq b.\,\texttt{withdraw}\,\mathsf{B}$$
$$+\,\texttt{after}\,(3n+2):\texttt{withdraw}\,\mathsf{B}\big)$$
$$+\,\texttt{after}\,(3n+1):\texttt{withdraw}\,\mathsf{A}$$

$$Split_\mathsf{A} = \texttt{split}\,(4 \rightarrow \texttt{withdraw}\,\mathsf{A} \mid 2 \rightarrow \texttt{withdraw}\,\mathsf{B})$$
$$Split_\mathsf{B} = \texttt{split}\,(4 \rightarrow \texttt{withdraw}\,\mathsf{B} \mid 2 \rightarrow \texttt{withdraw}\,\mathsf{A})$$

FIGURE 4. A recursive coin flipping game.

of A and B are equal, otherwise it is B. At this point, the contract can be renegotiated as $\mathsf{X_A}\langle 1\rangle$ or $\mathsf{X_B}\langle 1\rangle$, depending on the flip winner (the parameter 1 represents the round). If players do not agree on the renegotiation, then the funds are split fairly, according to the current expected win. The contract $\mathsf{X_A}\langle n\rangle$ requires A and B to generate fresh secrets for the n-th round. If A wins again, she can withdraw the pot, otherwise the contract can be renegotiated as $\mathsf{X_B}\langle n+1\rangle$. If the players do not agree on the renegotiation, the pot is split fairly between them. The contract $\mathsf{X_B}$ is similar.

This game is fair, i.e. the expected payoff of a rational player is always non-negative, notwithstanding the behaviour of the other player. Rational players must choose random secrets in $\{0,1\}$, since non uniformly distributed secrets can make the adversary bias the coin flip in her favour. Further, choosing a secret different from 0 or 1 would be irrational: if done by B, this would prevent himself from revealing (by the predicate in the $\texttt{reveal}\,b$), and so A could win after the timeout; if done by A, this would make B win the round (since B wins when the secrets are different). Rationality also requires to reveal secrets in time (before the alternative $\texttt{after}$ branch is enabled), and to take the $Split$ branch if restipulation does not occur in time. This ensures that, when renegotiation happens, there is still time to reveal the round secrets. Indeed, a late renegotiation could enable the other player to win by timeout. To show fairness, first consider the case where renegotiation always happens. A rational player wins each coin flip with probability $1/2$, at least: so, the probability of winning the whole game is also $1/2$, at least. In the general case, the renegotiation at the end of each round may fail. When this happens, the rational player takes the $Split$ branch, distributing the pot according to the expected payoff in the *current* game state, thus ensuring the fairness of the game. The player who won the last coin flip is expected to win p B̵, with $p = 1/2 \cdot 6 + 1/2 \cdot (1/2 \cdot p + 1/2 \cdot 0)$, giving $p = 4$. Accordingly,

the *Split* contracts transfer 4Ƀ to the winner of the last flip and $(6-4)\text{Ƀ} = 2\text{Ƀ}$ to the other player.

The contract *CFG* is liquid, because the `after` branches always offer liquidable alternatives to potentially blocking branches.

5. A SAFE ABSTRACTION OF THE BITML SEMANTICS

The concrete BitML semantics is infinite-state, because participants can always create new deposits, stipulate new contracts and renegotiate them, and can advance the current time. In this section we introduce an abstract semantics of BitML, which reduces the state space to a finite one, safely approximating liquidity. We construct our abstraction in three steps:

- First, we abstract concrete configurations Γ as *abstract configurations* $\alpha_{\mathsf{A},X}(\Gamma)$, where $\mathsf{A} \in \mathsf{Hon}$, and X is the (finite) set of contract names under observation. Roughly, $\alpha_{\mathsf{A},X}(\Gamma)$ discards all the terms in the configuration, except for the contracts X, which are abstracted as follows. We remove `reveal` actions, authorizations, and time constraints, only recording whether reducing a contract D requires cooperation from some participant different from A. Further, the abstraction discards the actual parameters $\mathcal{E}$ of renegotiations.
- We define a semantics $\rightarrow_\sharp$ of abstract configurations. This semantics partitions the moves in two sets: the moves x, which represent a reduction of the contract x that can be performed by A *alone*, and the moves $* : x$, which instead represent actions that may require cooperation from other participants. Intuitively, the moves of the first kind are those required by liquidability. Although this abstract semantics substantially simplifies the concrete one, it is still infinite-state. We establish that the abstract semantics is an over-approximation of the concrete one (Theorem 5.4). Further, we show that abstract runs containing only moves of A alone under-approximate concrete runs (Theorem 5.5).
- We define a finite-state refinement $\rightarrow_{\sharp fin}$ of $\rightarrow_\sharp$, which allows for model-checking liquidity. Liquidity w.r.t. $\rightarrow_{\sharp fin}$ is proved to be equivalent to liquidity w.r.t. $\rightarrow_\sharp$ (Theorem 5.10).

We start by defining the abstraction of configurations. Hereafter, we assume that all the abstractions are done w.r.t. the same honest participant A.

Definition 5.1 (Abstraction of configurations). For all sets of contract names X, we define the contract abstraction function α_{A} and the configuration abstraction function $\alpha_{\mathsf{A},X}$ in Figure 5. We abstract each defining equation $\mathtt{X}(\vec{\beta}) = \{G\}C$ as $\mathtt{X} =_\sharp \alpha_{\mathsf{A}}(C)$. The syntax of abstract configurations is implicitly given by the equations in Figure 5. We further allow recursion variables $\mathtt{X}$ within abstract configurations. We identify $* : * : D$ with $* : D$.

The equations in Figure 5 follow the intuition described above. The prefix τ used in `reveal` is used to abstract the fact that A alone can unconditionally reveal some of her secrets. Instead, we abstract as $* : \tau$ the case where some of the secrets to be revealed do not belong to A, or the truth of the predicate p is unknown. Since the secrets have been removed from configurations, this is a conservative (but safe) abstraction.

We now describe the abstract semantics of BitML. In the relation $\rightarrow_\sharp$ between abstract configurations, the rules to advertise contracts, for deposits, and for delays are removed. There is a rule for making a contract $\langle \mathtt{withdraw}\ \mathsf{A}, v \rangle_x$ reduce to a deposit $\langle \mathsf{A}, v \rangle_y$ is replaced so that $\langle \mathtt{withdraw}\ \mathsf{A}, v \rangle_x$ reduces to 0 (the empty configuration).

$$\alpha_{\mathsf{A}}\left(\sum_{i\in I} D_i\right) = \sum_{i\in I} \alpha_{\mathsf{A}}(D_i) \qquad \alpha_{\mathsf{A}}(\mathtt{withdraw\ B}) = \mathtt{withdraw\ B}$$

$$\alpha_{\mathsf{A}}(\mathtt{reveal}\ \vec{a}\ \mathtt{if}\ p.\ C) = \begin{cases} \tau.\,\alpha_{\mathsf{A}}(C) & \text{if } p = true \text{ and } \vec{a} \subseteq \mathcal{S}_{\mathsf{A}} \\ *:\tau.\,\alpha_{\mathsf{A}}(C) & \text{otherwise} \end{cases}$$

$$\alpha_{\mathsf{A}}\left(\mathtt{split}\ \left(\|_{i\in I} v_i \to C_i\right)\right) = \mathtt{split}\ \left(\|_{i\in I} \alpha_{\mathsf{A}}(C_i)\right)$$

$$\alpha_{\mathsf{A}}(\mathsf{B}:D) = \begin{cases} \alpha_{\mathsf{A}}(D) & \text{if } \mathsf{B} = \mathsf{A} \\ *:\alpha_{\mathsf{A}}(D) & \text{otherwise} \end{cases} \qquad \alpha_{\mathsf{A}}(\mathtt{after}\,t:D) = \alpha_{\mathsf{A}}(D)$$

$$\alpha_{\mathsf{A}}(*:\mathtt{rngt\ X}\langle\vec{\mathcal{E}}\rangle) = *:\mathtt{rngt\ X}$$

$$\alpha_{\mathsf{A},X}(\Delta \mid \Delta') = \alpha_{\mathsf{A},X}(\Delta) \mid \alpha_{\mathsf{A},X}(\Delta')$$

$$\alpha_{\mathsf{A},X}(\langle C, v\rangle_x) = \langle\alpha_{\mathsf{A}}(C)\rangle_x \quad \text{if } x \in X \qquad\qquad \alpha_{\mathsf{A},X}(\Delta) = 0 \quad \text{otherwise}$$

FIGURE 5. Abstraction of contracts and configurations.

$$\frac{}{\langle\mathtt{withdraw\ B}\rangle_x \mid \Gamma^\sharp \xrightarrow{x}_\sharp \Gamma^\sharp}\ {\scriptstyle[\text{A-Withdraw}]} \qquad \frac{y\ \text{fresh}}{\langle\tau.C\rangle_x \mid \Gamma^\sharp \xrightarrow{x}_\sharp \langle C\rangle_y \mid \Gamma^\sharp}\ {\scriptstyle[\text{A-Rev}]}$$

$$\frac{y_i\ \text{fresh}}{\langle\mathtt{split}\ \|_i\ C_i\rangle_x \mid \Gamma^\sharp \xrightarrow{x}_\sharp \|_i\ \langle C_i\rangle_{y_i} \mid \Gamma^\sharp}\ {\scriptstyle[\text{A-Split}]}$$

$$\frac{\langle D\rangle_x \mid \Gamma^\sharp \xrightarrow{\ell}_\sharp \Gamma'^\sharp \quad x \in \mathrm{cn}(\ell)}{\langle D + C\rangle_x \mid \Gamma^\sharp \xrightarrow{\ell}_\sharp \Gamma'^\sharp}\ {\scriptstyle[\text{A-Branch}]} \qquad \frac{\langle D\rangle_x \mid \Gamma^\sharp \xrightarrow{\ell}_\sharp \Gamma'^\sharp \quad x \in \mathrm{cn}(\ell)}{\langle *:D\rangle_x \mid \Gamma^\sharp \xrightarrow{*:\ell}_\sharp \Gamma'^\sharp}\ {\scriptstyle[\text{A-Ext}]}$$

$$\frac{\mathtt{X} =_\sharp C \quad y\ \text{fresh}}{\langle *:\mathtt{rngt\ X}\rangle_x \mid \Gamma^\sharp \xrightarrow{*:x}_\sharp \langle C\rangle_y \mid \mathtt{X} \mid \Gamma^\sharp}\ {\scriptstyle[\text{A-Rngt}]}$$

FIGURE 6. Abstract BitML semantics.

Definition 5.2 (Abstract semantics). We define the relation $\to_\sharp$ between abstract configurations in Figure 6. We use ℓ to range over labels, which have the form x or the form $*:x$, where x is a contract name. We identify $*:*:\ell$ with $*:\ell$. We define $\mathcal{L}^\sharp$ as the subset of labels of the form x. An *abstract run* $\mathcal{R}^\sharp$ is a sequence $\Gamma_0 \to_\sharp \Gamma_1 \to_\sharp \cdots$.

We briefly comment the rules in Figure 6, which define the abstract semantics $\to_\sharp$ of BitML. Most rules are straightforward. In rule [A-Ext], we record in the label $*:\ell$ the fact that reducing a contract $*:D$ might require the cooperation from some other participant. In rule [A-Rngt], we record through the term $\mathtt{X}$ in the configuration the fact that we have unfolded the recursion variable $\mathtt{X}$. This is not strictly needed to make the abstract semantics safely approximate the concrete one: rather, it is a technical expedient to build the finite-state semantics $\to_{\sharp fin}$ on-top of $\to_\sharp$.

Correspondence between the semantics. We now establish a correspondence between the abstract and the concrete semantics of BitML. In all the statements below, we assume that the concrete configuration Γ is reachable from an initial configuration.

We introduce below the notion of *descendants* of a set of contracts, which is dual to that of originator in Definition 3.1. This notion is exploited whenever we reduce an abstract configuration $\alpha_{A,X}(\Gamma)$ mimicking the steps of a concrete run $\mathcal{R}$ from Γ to Γ'. Intuitively, the resulting configuration will be an abstraction of Γ' against the descendants of X.

Definition 5.3 (Descendants). For all concrete configurations Γ, runs $\mathcal{R} = \Gamma \rightarrow_\flat^* \Gamma'$, and set of contract names X, we define the set of contract names $desc_\Gamma(\mathcal{R}, X)$ as follows:

$$desc_\Gamma(\mathcal{R}, X) \;=\; \big\{ x \in cn(\Gamma') \,\big|\, orig_\Gamma(\mathcal{R}, x) \in X \big\}$$

Theorem 5.4 (Over-approximation). *Let $X \subseteq cn(\Gamma)$, and let $\mathcal{R} = \Gamma \rightarrow_\flat^* \Gamma'$ be a concrete run. Then, there exist $X_1, X_2, \ldots$ such that:*

$$\alpha_{A,X}(\Gamma) \rightarrow_\sharp^* \alpha_{A,desc_\Gamma(\mathcal{R},X)}(\Gamma') \mid \;\|_i X_i$$

Proof. (sketch) By induction on the number of concrete moves. We check that each move $\Delta \rightarrow_\flat \Delta'$ is matched by zero or one abstract moves. First, note that the abstract configuration contains only contracts in X. We have the following cases, according to the concrete move:

- If $\Delta \rightarrow_\flat \Delta'$ does neither consume nor create a contract, then the abstractions of Δ and Δ' are the same, so we match the concrete move with zero abstract moves.
- If $\Delta \rightarrow_\flat \Delta'$ creates contracts without consuming any contract, then it is an instance of rule [C-Init]. In this case, the created contract is not a descendant of any contract in X, so it does not occur in the abstraction of Δ'. As in the previous case, we match the concrete move with zero abstract moves.
- If $\Delta \rightarrow_\flat \Delta'$ consumes a contract $x \notin X$, then the abstraction discards the contract x and its descendants, and so we match the concrete move with zero abstract moves.
- If $\Delta \rightarrow_\flat \Delta'$ consumes a contract $x \in X$, then one of the rules [C-Split], [C-Rev], [C-Withdraw] or [C-Rngt] have been used. The moves [C-Split], [C-Withdraw] and [C-Rngt] are matched by the corresponding [A-*] abstract moves. The move [C-Rev] is matched either by [A-Rev] or [A-Ext], depending on how the `reveal` was abstracted. Note that [C-Rngt] may instantiate the formal parameters in the contract with actual values. However, these values are abstracted away by α_A, so this instantiation is immaterial. The term X introduced by [A-Rngt] is accounted for by the statement (it will belong to $\|_i X_i$). □

Theorem 5.5 (Under-approximation). *Let $X \subseteq cn(\Gamma)$, and assume that:*

$$\alpha_{A,X}(\Gamma) \mid \;\|_i X_i \; \rightarrow_\sharp^* \; \Gamma'_\sharp \qquad \text{with labels in } \mathcal{L}^\sharp$$

Then, there exist $\mathcal{R}$ and Γ' such that:

$$\mathcal{R} = \Gamma \; \rightarrow_\flat^* \; \Gamma' \qquad\qquad \text{with labels in } \mathcal{L}_A^\flat, \text{ and } \Gamma'_\sharp = \alpha_{A,desc_\Gamma(\mathcal{R},X)}(\Gamma') \mid \;\|_i X_i$$

Proof. (sketch) By induction on the number of moves. Since each abstract move must be in $\mathcal{L}^\sharp$, it must be derived by one of the rules [A-Withdraw], [A-Rev], or [A-Split], possibly as a premise of [A-Branch]. To fall into these cases, the concrete contract must have one of the following forms:

- `withdraw B` or `split` $\cdots$. Here, the abstract move is matched by the concrete moves [C-Withdraw] or [C-Split], the labels of which belong to $\mathcal{L}_A^\flat$.

- `reveal` $\vec{a}$ `if` *true* where $\vec{a}$ are secrets of A, the participant w.r.t. whom we abstract. In this case, the abstract move [A-Rev] is matched by [C-Rev], the label of which belongs to $\mathcal{L}_A^\flat$. If the secrets have not been revealed, yet, we perform one or more [C-AuthRev] moves (the labels of which also belong to $\mathcal{L}_A^\flat$) before the [C-Rev] move. Note that the premise $N \neq \bot$ in rule [C-AuthRev] is satisfied, because we are assuming that the participant A against whom we are abstracting is honest (A $\in$ Hon), and rule [C-AuthCommit] ensures that honest participants have committed to secrets $N \neq \bot$.

- any of the above, constrained by `after` $t : \cdots$ or B $: \cdots$ with B $=$ A. Here, if needed, we perform a [C-Delay] to advance the current time until time t (a move in $\mathcal{L}_A^\flat$). Then, if needed, A can perform a [C-AuthBranch] to authorize the branch (this move is also in $\mathcal{L}_A^\flat$). After that, the constraints are satisfied, so we proceed as in the previous items. $\square$

Safe approximation of liquidity. Theorem 5.7 below establishes the soundness of our abstraction w.r.t. liquidity. It exploits the following corollary for liquidability.

Corollary 5.6 (Abstraction soundness against liquidability). *Let $x \in X \subseteq \mathrm{cn}(\Gamma)$. If x is liquidable in $\alpha_{A,X}(\Gamma) \mid \vec{Y}$ w.r.t. $\rightarrow_\sharp / \mathcal{L}^\sharp$, then x is liquidable in Γ w.r.t. $\rightarrow_\flat / \mathcal{L}_A^\flat$.*

Proof. Since x is liquidable in $\alpha_{A,X}(\Gamma) \mid \vec{Y}$, there exists an abstract run with labels in $\mathcal{L}^\sharp$ and leading to a configuration $\Gamma'^\sharp$ without contracts originated from x. By Theorem 5.5, there is a corresponding concrete run with labels in $\mathcal{L}_A^\flat$ starting from Γ and leading to a configuration Γ' the abstraction of which is $\Gamma'^\sharp$. Since the abstraction preserves all the contracts originated from x, in Γ' there are none. Therefore, x is liquidable in Γ. $\square$

Theorem 5.7 (Abstraction soundness against liquidity). *Let $x \in X \subseteq \mathrm{cn}(\Gamma)$. If x is liquid in $\alpha_{A,X}(\Gamma)$ w.r.t. $\rightarrow_\sharp / \mathcal{L}^\sharp$, then x is liquid in Γ w.r.t. $\rightarrow_\flat / \mathcal{L}_A^\flat$.*

Proof. Let $\mathcal{R} = \Gamma \rightarrow \cdots \rightarrow \Gamma'$ be a concrete run, and let y be such that $orig_\Gamma(\mathcal{R}, y) = x$. By Theorem 5.4, there exists a corresponding abstract run:

$$\Gamma^\sharp = \alpha_{A,X}(\Gamma) \rightarrow_\sharp^* \Gamma'^\sharp = \alpha_{A,desc_\Gamma(\mathcal{R},X)}(\Gamma') \mid \|_i X_i$$

Since x is liquid in $\Gamma^\sharp$ and y is a descendant of x in the abstract run, y is liquidable in $\Gamma'^\sharp$. By Corollary 5.6, y is liquidable in Γ'. Therefore, x is liquid in Γ. $\square$

A finite-state abstraction of BitML. The abstract semantics $\rightarrow_\sharp$ is infinite-state, even up-to renaming of contract names. Indeed, each application of rule [A-Rngt] unfolds a contract X, which can spawn other parallel contracts before recursing. Therefore, the number of parallel contracts in reachable abstract configurations can grow unboundedly. This hinders verification based on model-checking the whole state space.

We now introduce another abstract semantics, called $\rightarrow_{\sharp fin}$, which has a *finite* number of reachable configurations (up-to renaming of contract names), and which preserves the liquidity w.r.t. $\rightarrow_\sharp$: more specifically, a contract is liquid w.r.t. $\rightarrow_\sharp$ if and only if it is liquid w.r.t. $\rightarrow_{\sharp fin}$. We define the transition relation $\rightarrow_{\sharp fin}$ by the following rule:

$$\frac{\Gamma \xrightarrow{\ell}_\sharp \Gamma' \quad \nexists \Delta, X : \Gamma' = \Delta \mid X \mid X}{\Gamma \xrightarrow{\ell}_{\sharp fin} \Gamma'}$$

Intuitively, $\to_{\sharp fin}$ can mimic any move of $\to_\sharp$, except for the moves [A-Rngt] which renegotiate a contract which has already been renegotiated. Technically, this is ensured by constraining the configuration to contain at most one occurrence of each recursion variable $\mathtt{X}$.

Theorem 5.8. *Starting from any abstract configuration, the set of states reachable through $\to_{\sharp fin}$ is finite, up-to renaming of contract names.*

Proof. Direct consequence of the definition of the abstract semantics. $\square$

The following lemma establishes the equivalence of $\to_\sharp$ and $\to_{\sharp fin}$ against liquidability, and it is instrumental to prove that they also agree on liquidity (Theorem 5.10).

Lemma 5.9. *Let $\Gamma^\sharp$ be an abstract configuration without multiple occurrences of any $\mathtt{X}$. Then, for all contract names x:*

$$x \text{ liquidable in } \Gamma^\sharp \text{ w.r.t. } \to_\sharp /\mathcal{L}^\sharp \iff x \text{ liquidable in } \Gamma^\sharp \text{ w.r.t. } \to_{\sharp fin} /\mathcal{L}^\sharp$$

Proof. Recall that liquidability only considers moves in $\mathcal{L}^\sharp$. The only difference between $\to_\sharp$ and $\to_{\sharp fin}$ is that the latter forbids, in some configurations, the application of rule [A-Rngt]. Since the label of this rule is not in $\mathcal{L}^\sharp$, this difference is immaterial for liquidability. $\square$

Theorem 5.10. *Let $\Gamma^\sharp$ be an abstract configuration without occurrences of any $\mathtt{X}$. Then, for all contract names x:*

$$x \text{ liquid in } \Gamma^\sharp \text{ w.r.t. } \to_\sharp /\mathcal{L}^\sharp \iff x \text{ liquid in } \Gamma^\sharp \text{ w.r.t. } \to_{\sharp fin} /\mathcal{L}^\sharp$$

Proof. (sketch) For $\Rightarrow$, consider a run $\mathcal{R}^\sharp$ of $\to_{\sharp fin}$ starting from $\Gamma^\sharp$ and leading to some $\Gamma'^\sharp$, and let y be a descendant of x. Since $\to_\sharp$ includes $\to_{\sharp fin}$, $\mathcal{R}^\sharp$ is also a run of $\to_\sharp$. Since x is liquid in $\Gamma^\sharp$ w.r.t. $\to_\sharp$, then y must be liquidable in $\Gamma'^\sharp$ w.r.t. $\to_\sharp$. By Lemma 5.9, it follows that y is liquidable also in $\to_{\sharp fin}$. Therefore, x is liquid in $\Gamma^\sharp$ w.r.t. $\to_{\sharp fin}$.

For $\Leftarrow$, take any contract $\langle C \rangle_y$ which originates from the contract x in a run of $\to_\sharp$ starting from $\Gamma^\sharp$. By contradiction, assume that y is not liquidable for $\to_\sharp$. In the $\to_\sharp$ run, $\langle C \rangle_y$ might be reached after several renegotiations involving some defined contracts $\mathtt{X}_1, \mathtt{X}_2, \ldots$ more than once. In such case, it is also possible to reach C without renegotiating the same $\mathtt{X}_i$ more than once, because each use of rule [A-Rngt] on $\mathtt{X}_i$ spawns the same contract, and contracts in parallel do not interact. Hence, there exists some $\to_\sharp$ run leading to a non liquidable $\langle C \rangle_z$ and whose configurations never include any $\mathtt{X}_i$ more than once. This run is therefore also a $\to_{\sharp fin}$ run, leading to a non liquidable z according to Lemma 5.9 — contradiction with the liquidity of x w.r.t. $\to_{\sharp fin}$. $\square$

6. The BitML toolchain

We now describe the BitML toolchain, whose architecture is displayed in Figure 7. The development workflow is the following: (a) write the BitML contract in the DrRacket IDE; (b) verify that the contract is liquid w.r.t. the given participant; (c) compile the contract to Bitcoin transactions; (d) execute the contract, by appending these transactions to the Bitcoin blockchain according to the chosen strategy. The verifier implements the abstract BitML semantics in Maude, a model-checking framework based on rewriting
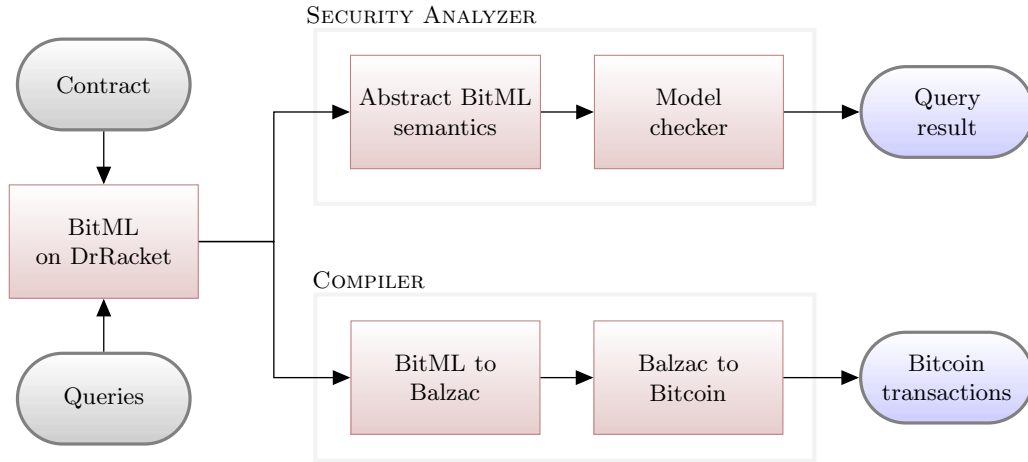
FIGURE 7. Toolchain architecture.

logic [CDE⁺02]. The toolchain is open-source[2]; a tutorial is available online[3], including references to our experiments on the Bitcoin testnet.

To evaluate our toolchain, we use a benchmark of representative use cases[4] including financial contracts [ST18, BKT17], auctions, lotteries [ADMM16, MB17] and gambling games. For each contract in the benchmark, we display in Table 1 the number N of involved participants, the number T of transactions obtained by the compiler, and the number of Maude rewrites R for checking liquidity. Notably, the tool automatically verifies that all the contracts in the benchmark are liquid. The verification time for all the benchmarks is in the order of milliseconds on a consumer-grade laptop, except for *ZCB3*, which requires $\sim 1s$.

We compare the performance of our tool against [ADMM14b], which models Bitcoin contracts in Uppaal, a model-checking framework based on Timed Automata. The most complex contract modelled in [ADMM14b] is the mutual timed commitment with 2 participants: this requires $\sim 30s$ to be verified in Uppaal, while our tool verifies the same property in $\sim 1ms$. This speedup is due to the higher abstraction level of BitML over [ADMM14b], which operates at the (lower) level of Bitcoin transactions.

One of the main difficulties that we have encountered in developing contracts is that some complex BitML specifications can not be compiled to pure Bitcoin, because of the 520-byte limit on the size of each value pushed to the evaluation stack [btc19]. In some cases, we managed to massage the BitML contract so to make its compilation respect the constraint. For instance, a pattern that easily violates the constraint is the following:

```
(choice (revealif (b) (pred (p0)) (C0))
        (revealif (b) (pred (p1)) (C1))
        (after T       (C2)))
```

The `choice` is compiled into a transaction whose redeem script encodes the disjunction of *three* logical conditions, corresponding to the three branches of the `choice`. Depending

---

[2]https://github.com/bitml-lang
[3]https://blockchain.unica.it/bitml
[4]https://github.com/bitml-lang/bitml-compiler/tree/master/examples/benchmarks

Contract	N	T	R
Mutual timed commitment	2	15	72
Mutual timed commitment	3	34	207
Mutual timed commitment	4	75	644
Mutual timed commitment	5	164	2097
Escrow (early fees)	3	12	104
Escrow (late fees)	3	11	110
Zero Coupon Bond	3	8	189
Coupon Bond	3	18	9101
Future(C)	3	$5 + T_C$	$136 + R_C$
Option(C, D)	3	$14 + T_C + T_D$	$162 + R_C + R_D$
Lottery ($O(N^2)$ collateral)	2	15	1466
Lottery (0 collateral)	2	8	31
Lottery (0 collateral)	4	587	167
Rock-Paper-Scissors	2	23	2322
Morra game	2	40	89
Shell game	2	23	48
Auction (2 turns)	2	42	218
Coin flipping game	2	32	563
Zero coupon bond (v3)	3	44	1196813

TABLE 1. Benchmarks for the BitML toolchain.

on the predicates `p0` and `p1`, and on the number of participants in the contract, this script may violate the 520-byte constraint. A workaround is to rewrite the pattern above into:

```
(choice (revealif (b) (pred (p0)) (C0))
        (after T (tau (choice
                      (revealif (b) (pred (p1)) (C1))
                      (after T1    (C2))))))
```

In this case the compilation includes two transactions, corresponding to the two `choices`. The scripts of these transactions encode the disjunction of *two* logical conditions, corresponding to the two branches of the `choices`. Using this workaround we have managed to compile a 4-players lottery into standard transactions, at the price of increasing the number of transactions (587 for the standard version *vs.* 138 for the nonstandard one). Similar techniques (e.g. simplification of predicates) allowed us to compile all the contracts in Table 1 into standard Bitcoin transactions.

In general, the 520-byte constraint intrinsically limits the expressiveness of Bitcoin contracts: for instance, since public keys are 33 bytes long, a contract which needs to simultaneously verify 15 signatures can not be implemented using standard transactions.

7. VARIANTS OF LIQUIDITY

We now discuss some variants of the notion of liquidity of Section 3.

Liquidity under a strategy. The runs $\mathcal{R} = \Gamma \to \cdots \to \Gamma'$ in Definition 3.3 allow any participant to perform *any* enabled move. For instance, consider the contract:

$$\{A\!:\!1 \,@\, x \mid A\!:\!\texttt{secret}\, a \mid A\!:\!\texttt{secret}\, a'\}(\texttt{reveal}\, a.\texttt{withdraw}\, A + \texttt{reveal}\, a'.B\!:\!\texttt{withdraw}\, A)$$

This contract is *not* liquid for A: indeed, if A performs `reveal` a', the reduct B : `withdraw` A is not liquidable by A alone, since B may refuse to give his authorization. To overcome this issue, A can follow the strategy of always performing the `reveal` a. In this way, she is sure to be able liquidate the contract from any reachable state. A possible extension of the notion of liquidity in Definition 3.3 is to make it parametric on A's strategy, and consider only the runs $\mathcal{R} = \Gamma \to \cdots \to \Gamma'$ which are coherent with it.

Multiparty liquidity. Definition 3.3 requires that A *alone* can liquidate each descendent of the contract. We can relax it by considering a set of collaborative participants. For instance, consider an escrow contract between A and B, involving also a mediator M:

$$Escrow = \mathsf{A : withdraw\ B} \ + \ \mathsf{B : withdraw\ A} \ + \ \mathsf{A} : Resolve \ + \ \mathsf{B} : Resolve$$

$$Resolve = \mathtt{split}(0.1 \to \mathtt{withdraw\ M}$$
$$| \ 0.9 \to \mathsf{M : withdraw\ A} + \mathsf{M : withdraw\ B})$$

After the contract has been stipulated, A can choose to pay B, by authorizing the first branch. Similarly, B can allow A to take her money back, by authorizing the second branch. If they do not agree, any of them can invoke a mediator M to resolve the dispute, invoking a *Resolve* branch. There, the initial deposit (say, of vḂ) is split in two parts: $0.1v$ goes to the mediator, while $0.9v$ is assigned either to A and B, depending on M's choice. This contract is *not* liquid for A, because B can invoke the mediator, who can refuse to act, freezing the funds within the contract (similarly for B). Instead, assuming that M is collaborative, the contract is liquid for both A and B. Indeed, a collaborative M will always authorize either the `withdraw` A or the `withdraw` B to unlock $0.9v$. Multiparty liquidity where all participants are collaborative was used e.g. in [TDD+18] in the context of Ethereum contracts.

Quantitative liquidity. Definition 3.3 requires that no funds remain frozen within the contract. However, in some cases A could accept the fact that a portion of the funds remain frozen, especially when these funds would be assigned to other participants. We could define a contract v-*liquid* for A if at least v bitcoins are guaranteed to be redeemable by anyone. For instance, *Lottery*(*Win*) of Section 4 is non-liquid for A, but it is 4Ḃ-liquid. Instead, *Lottery*(*Win₂*) is 6Ḃ-liquid, and then also liquid, under this strategy. A refinement of this notion could require that at least vḂ are transferred to A, rather than to any participant. Under this notion, both *Lottery*(*Win*) and *Lottery*(*Win₂*) would be 2Ḃ-liquid for A.

Other variants of liquidity may take into account the time when funds become liquid, the payoff of strategies (e.g., ruling out irrational adversaries), or fairness issues.

8. Variants of contract primitives

The renegotiation primitive we have proposed for BitML is motivated by its simplicity, and by the possibility of compiling into standard Bitcoin transactions. By adding some degree of complexity, we can devise more general primitives, which could be useful in certain scenarios. We discuss below some alternatives.

Renegotiation-time parameters. The primitive $* : \mathtt{rngt}\ \mathtt{X}\langle\vec{\mathcal{E}}\rangle$ allows participants to choose at run-time only the deposit variables used in the renegotiated contracts, and to commit to new secrets. A possible extension is to allow participants to choose at run-time *arbitrary* values for the renegotiation parameters $\vec{\mathcal{E}}$.

For instance, consider a mortgage payment, where a buyer A must pay $10\ddot{\mathtt{B}}$ to a bank B in 10 installments. After A has paid the first five installments (of $1\ddot{\mathtt{B}}$ each), the bank might propose to renegotiate the contract, varying the amount of the installment. Using the BitML renegotiation primitive presented in Section 2, we could not model this contract, since the new amount and the number of installments are unknown at the time of the original stipulation. Technically, the issue is that the primitive $* : \mathtt{rngt}\ \mathtt{X}\langle\vec{\mathcal{E}}\rangle$ only involves static expressions $\mathcal{E}$, the value of which is determined at stipulation time.

To cope with non-statically known values, we could extend guarded contracts with terms of the form $* : \mathtt{rngt}\ \mathtt{X}\langle\mathsf{B} : v\rangle$, declaring that the value v is to be chosen by B at renegotiation time. For instance, this would allow to model our installments payment plan as $\mathtt{IPP}\langle 1\rangle$, with the following defining equations:

$$\mathtt{IPP}\langle\beta < 5\rangle = \{\mathsf{A}\!:\!1\,@\,d\}\big(\mathtt{split}\ 1 \to \mathtt{withdraw}\ \mathsf{B}\ |\ 0 \to *:\mathtt{rngt}\ \mathtt{IPP}\langle\beta + 1\rangle\big)$$

$$\mathtt{IPP}\langle 5\rangle = \{\mathsf{A}\!:\!1\,@\,d\}\big(\mathtt{split}\ 1 \to \mathtt{withdraw}\ \mathsf{B}\ |\ 0 \to *:\mathtt{rngt}\ \mathtt{Y}\langle\mathsf{B} : k, \mathsf{B} : v\rangle\big)$$

$$\mathtt{Y}\langle\beta \neq 1, \beta'\rangle = \{\mathsf{A}\!:\!\beta'\,@\,d\}\big(\mathtt{split}\ \beta' \to \mathtt{withdraw}\ \mathsf{B}\ |\ 0 \to *:\mathtt{rngt}\ \mathtt{Y}\langle\beta - 1, \beta'\rangle\big)$$

$$\mathtt{Y}\langle 1, \beta'\rangle = \{\mathsf{A}\!:\!\beta'\,@\,d\}\ \mathtt{withdraw}\ \mathsf{B}$$

where in $\mathtt{IPP}\langle 5\rangle$, the bank chooses the number of installments k, as well as the amount v of each installment. Note that if A does not agree with these values, the renegotiation fails. A more refined version of the contract should take this possibility into account, by adding suitable compensation branches. Although adding the new primitive would moderately increase the complexity of the semantics and of the compiler, this extension can still be implemented on top of standard Bitcoin.

Renegotiation with a given set of participants. As we have remarked in Section 2, a renegotiation can be performed only if *all* the participants of the contract agree. We could relax this, by just requiring the agreement of a *given* set of participants (possibly, not among those who originally stipulated the contract).

For instance, consider an escrow service between a buyer A and a seller B for the purchase of an item worth $1\ddot{\mathtt{B}}$. The normal case is when the A authorizes the transfer of $1\ddot{\mathtt{B}}$ after receiving the item, but it may happen that a dishonest B never ships the item, or that a dishonest buyer never authorizes the payment. To cope with these cases, the participants could renegotiate the contract, including an escrow service M which mediates the dispute:

$$\mathsf{A} : \mathtt{withdraw}\ \mathsf{B}\ +\ \mathsf{B} : \mathtt{withdraw}\ \mathsf{A}\ +\ \mathsf{A} : \mathsf{M} : \mathtt{rngt}\ \mathtt{Refd}_\mathsf{A}\ +\ \mathsf{B} : \mathsf{M} : \mathtt{rngt}\ \mathtt{Refd}_\mathsf{B}$$

$$\mathtt{Refd}_\mathsf{P} = \{\mathsf{P}\!:\!0.1\,@\,d\}\ \mathtt{split}\ \big(0.1 \to \mathtt{withdraw}\ \mathsf{M}\ |\ 1 \to \mathtt{withdraw}\ \mathsf{P}\big)$$

where $\mathsf{A} : \mathsf{M} : \mathtt{rngt}\ \mathtt{Refd}_\mathsf{A}$ means that only A and M need to agree in order for the contract $\mathtt{Refd}_\mathsf{A}$ to be executed, resolving the dispute. In this case it is crucial that the renegotiation is possible even without the agreement between A and B. Indeed, if M decides to refund A (by authorizing $\mathtt{Refd}_\mathsf{A}$), it is not to be expected that also B agrees. Similarly to the one discussed before, also this extension can be implemented on-top of Bitcoin.

Non-consensual renegotiation. In the variants of $*:\mathtt{rngt}$ discussed before, renegotiation requires one or more participants to agree. Hence, each use of $*:\mathtt{rngt}$ must include suitable alternative branches, to be fired in case the renegotiation fails. In certain scenarios, we may want to renegotiate the contract without the participants having to agree. To this purpose, we can introduce a new primitive $\mathtt{call\ X}$, which continues as $\mathtt{X}$ without requiring anyone to agree. We assume that the defining equations of this primitive have the form $\mathtt{X}(\vec{\beta}) = \{v\}C$, where v represents the amount of ฿ added to the contract, by anyone.

We exemplify the new primitive in a two-players game which starts with a bet of 1฿ from A, and a bet of 2฿ from B. Then, starting from A, players take turns adding 2฿ each to the pot. The first one who is not able to provide the additional 2฿ within a given time loses the game, allowing the other player to take the whole pot. The contract is as follows:

$$C = \{\mathsf{A}\!:\!1\,@\,x \mid \mathsf{B}\!:\!2\,@\,y\}(\mathtt{call\ X_A}\langle 2\rangle + \mathtt{after}\,1\!:\!\mathtt{withdraw\ B})$$

$$\mathtt{X_A}\langle n\rangle = \{2\}(\mathtt{call\ X_B}\langle n+1\rangle + \mathtt{after}\,n\!:\!\mathtt{withdraw\ A})$$

$$\mathtt{X_B}\langle n\rangle = \{2\}(\mathtt{call\ X_A}\langle n+1\rangle + \mathtt{after}\,n\!:\!\mathtt{withdraw\ B})$$

Unlike $*:\mathtt{rngt}$, the action $\mathtt{call}$ can be fired without the authorizations of all the players: it just requires that the authorization to gather 2฿ is provided, by anyone. Even though the sender of these 2฿ is not specified in the contract, it is implicit in the game mechanism: for instance, when $\mathtt{X_A}\langle n\rangle$ calls $\mathtt{X_B}\langle n+1\rangle$, only participant B is incentivized to add 2฿, since not doing so will make A win.

Implementing the $\mathtt{call}$ primitive on top of Bitcoin seems unfeasible: even if it were possible to use complex off-chain multiparty computation protocols [GMR^{+}19], doing so might be impractical. Rather, we would like to extend Bitcoin as much as needed for the new primitive. In our implementation of BitML, we compile contracts to sets of transactions and make participants sign them. In standard BitML this is doable since, at stipulation time, we can finitely over-approximate the reducts of the original contract. Recursion can make this set infinite, e.g. $\mathtt{X_A}\langle 2\rangle, \mathtt{X_A}\langle 3\rangle, \ldots$, hence impossible to compile and sign statically. A way to cope with this is to extend Bitcoin with *malleable* signatures which only cover the part of the transaction not affected by the parameter n in $\mathtt{X_B}\langle n\rangle$. Further, signatures must not cover the in fields of transactions, since they change as recursion unfolds. In this way, the same signature can be reused for each call.

Adding malleability provides flexibility, but poses some risks. For instance, instead of redeeming the transaction corresponding to $\mathtt{X_A}\langle n\rangle$ with the transaction of $\mathtt{X_B}\langle n+1\rangle$ one could instead use the transaction of $\mathtt{X_B}\langle n+100\rangle$, since the two transactions have the same signature. To overcome this problem, we could add a new opcode to allow the output script of $\mathtt{X_B}\langle n\rangle$ to access the parameter in the redeeming transaction, so to verify that it is indeed $n+1$ as intended. Similarly, to check that we have 2฿ more in the new transaction, an opcode could provide the value of the new output. The same goal could be achieved by exploiting *covenants* [MES16, OP17, BLZ20].

9. CONCLUSIONS

We have investigated linguistic primitives to renegotiate BitML contracts, and their implementation on standard Bitcoin. More expressive primitives could be devised by relaxing this constraint, e.g. assuming the extended UTXO model [CCM^{+}20].

Our verification technique is based on a sound abstraction of the state space of contracts. Since this abstraction is finite-state, it can be model-checked to verify the required

properties. If we assume that integers are unbounded, and that participants always accept renegotiations, the extension of BitML presented in Section 2 can simulate a counter machine, so making BitML Turing-complete. Hence, any verification technique for BitML cannot be sound and complete. Alternative techniques to model checking (e.g., type-based approaches [DBHP19]) could be used to analyse relevant contract properties.

References

[ABC17]   Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *Principles of Security and Trust (POST)*, volume 10204 of *LNCS*, pages 164–186. Springer, 2017.

[ABC⁺18]   Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli, Stefano Lande, and Roberto Zunino. SoK: unraveling Bitcoin smart contracts. In *POST*, volume 10804 of *LNCS*, pages 217–242. Springer, 2018.

[ABL⁺19]   Nicola Atzei, Massimo Bartoletti, Stefano Lande, Nobuko Yoshida, and Roberto Zunino. Developing secure Bitcoin contracts with BitML. In *ESEC/FSE*, 2019.

[ADMM14a]   Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Fair two-party computations via Bitcoin deposits. In *Financial Cryptography Workshops*, volume 8438 of *LNCS*, pages 105–121. Springer, 2014.

[ADMM14b]   Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. Modeling Bitcoin contracts by timed automata. In *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 8711 of *LNCS*, pages 7–22. Springer, 2014.

[ADMM14c]   Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on Bitcoin. In *IEEE S & P*, pages 443–458, 2014. First appeared on Cryptology ePrint Archive, http://eprint.iacr.org/2013/784.

[ADMM16]   Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on Bitcoin. *Commun. ACM*, 59(4):76–84, 2016.

[BDM16]   Waclaw Banasik, Stefan Dziembowski, and Daniel Malinowski. Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In *ESORICS*, volume 9879 of *LNCS*, pages 261–280. Springer, 2016.

[BK14]   Iddo Bentov and Ranjit Kumaresan. How to use Bitcoin to design fair protocols. In *CRYPTO*, volume 8617 of *LNCS*, pages 421–439. Springer, 2014.

[BKT17]   Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. Findel: Secure derivative contracts for Ethereum. In *Financial Cryptography Workshops*, volume 10323 of *LNCS*, pages 453–467. Springer, 2017.

[BLZ20]   Massimo Bartoletti, Stefano Lande, and Roberto Zunino. Bitcoin covenants unchained. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA*, volume 12478 of *ISOLA*, pages 25–42. Springer, 2020.

[BMZ20]   Massimo Bartoletti, Maurizio Murgia, and Roberto Zunino. Renegotiation and recursion in Bitcoin contracts. In *Proc. COORDINATION*, volume 12134 of *LNCS*, pages 261–278. Springer, 2020.

[btc19]   Bitcoin script size limit. https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki#520-byte-limitation-on-s 2019.

[BZ17]   Massimo Bartoletti and Roberto Zunino. Constant-deposit multiparty lotteries on Bitcoin. In *Financial Cryptography Workshops*, volume 10323 of *LNCS*. Springer, 2017.

[BZ18]   Massimo Bartoletti and Roberto Zunino. BitML: a calculus for Bitcoin smart contracts. In *ACM CCS*, 2018.

[BZ19]   Massimo Bartoletti and Roberto Zunino. Verifying liquidity of bitcoin contracts. In *POST*, volume 11426 of *LNCS*. Springer, 2019.

[CCM+20]   Manuel M.T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended UTXO model. In *Workshop on Trusted Smart Contracts*, 2020.

[CDE+02]   Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.

[DBHP19]   Ankush Das, Stephanie Balzer, Jan Hoffmann, and Frank Pfenning. Resource-aware session types for digital contracts. *CoRR*, abs/1902.06056, 2019.

[Fla12]   Matthew Flatt. Creating languages in Racket. *Commun. ACM*, 55(1):48–56, 2012.

[GMR+19]   Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. Sok: Off the chain transactions. *IACR Cryptology ePrint Archive*, 2019:360, 2019.

[JES00]   Simon L. Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering, functional pearl. In *International Conference on Functional Programming (ICFP)*, pages 280–292, 2000.

[KB14]   Ranjit Kumaresan and Iddo Bentov. How to use Bitcoin to incentivize correct computations. In *ACM CCS*, pages 30–41, 2014.

[KB16]   Ranjit Kumaresan and Iddo Bentov. Amortizing secure computation with penalties. In *ACM CCS*, pages 418–429, 2016.

[KMB15]   Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use Bitcoin to play decentralized poker. In *ACM CCS*, pages 195–206, 2015.

[KVV16]   Ranjit Kumaresan, Vinod Vaikuntanathan, and Prashant Nalini Vasudevan. Improvements to secure computation with penalties. In *ACM CCS*, pages 406–417, 2016.

[LCO+16]   Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *ACM CCS*, pages 254–269, 2016.

[MB17]   Andrew Miller and Iddo Bentov. Zero-collateral lotteries in Bitcoin and Ethereum. In *EuroS&P Workshops*, pages 4–13, 2017.

[MES16]   Malte Möser, Ittay Eyal, and Emin Gün Sirer. Bitcoin covenants. In *Financial Cryptography Workshops*, volume 9604 of *LNCS*, pages 126–141. Springer, 2016.

[OP17]   Russell O'Connor and Marta Piekarska. Enhancing Bitcoin transactions with covenants. In *Financial Cryptography Workshops*, volume 10323 of *LNCS*. Springer, 2017.

[ST18]   Pablo Lamela Seijas and Simon J. Thompson. Marlowe: Financial contracts on blockchain. In *ISoLA*, volume 11247 of *LNCS*, pages 356–375. Springer, 2018.

[TDD+18]   Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical Security Analysis of Smart Contracts. In *ACM CCS*, pages 67–82, 2018.

Appendix A. Semantics of Bitcoin deposits

We describe below the rules which deal with deposits, without directly affecting contracts.

The rule [Dep-AuthJoin] allows A to authorize the merge of two deposits x, y into a single one, creating the needed authorization. The label of the form $A : \cdots$ records that only participant A can perform this move. The rule [Dep-Join] uses this authorization to create a single deposit z of A.

The rules [Dep-AuthDivide] and [Dep-Divide] act similarly, allowing a deposit of A to be divided in two parts.

The rules [Dep-AuthDonate] and [Dep-Donate] allow A to transfer one of her deposits to another participant.

The pair of rules [Dep-AuthDestroy] and [Dep-Destroy] allow a set of participants to destroy a set of deposits $x_1 \cdots x_n$. To do that, first each participant A_i must provide the needed authorization $A_i[\vec{x}, i \rhd y]$ for their own deposit x_i. When all the authorizations have been collected, rule [Dep-Destroy] eliminates the deposits. The last pair of rules is needed to properly represent the fact that computational participants can create (and put on the ledger) transactions which do not have a counterpart in our symbolic model. To achieve a meaningful correspondence between the symbolic and the computational models, putting on the ledger such transactions is rendered with the rule [Dep-destroy].

$$\frac{}{\langle A, v \rangle_x \mid \langle A, v' \rangle_y \mid \Gamma \xrightarrow{A:x,y}_\flat \langle A, v \rangle_x \mid \langle A, v' \rangle_y \mid A[x, y \rhd \langle A, v + v' \rangle] \mid \Gamma} \text{[Dep-AuthJoin]}$$

$$\frac{\Gamma = A[x, y \rhd \langle A, v + v' \rangle] \mid A[y, x \rhd \langle A, v + v' \rangle] \mid \Gamma' \quad z \text{ fresh}}{\langle A, v \rangle_x \mid \langle A, v' \rangle_y \mid \Gamma \xrightarrow{join(x,y)}_\flat \langle A, v + v' \rangle_z \mid \Gamma'} \text{[Dep-Join]}$$

$$\frac{}{\langle A, v + v' \rangle_x \mid \Gamma \xrightarrow{A:x,v,v'}_\flat \langle A, v + v' \rangle_x \mid A[x \rhd \langle A, v \rangle, \langle A, v' \rangle] \mid \Gamma} \text{[Dep-AuthDivide]}$$

$$\frac{\Gamma = A[x \rhd \langle A, v \rangle, \langle A, v' \rangle] \mid \Gamma' \quad y, y' \text{ fresh}}{\langle A, v + v' \rangle_x \mid \Gamma \xrightarrow{divide(x,v,v')}_\flat \langle A, v \rangle_y \mid \langle A, v' \rangle_{y'} \mid \Gamma'} \text{[Dep-Divide]}$$

$$\frac{}{\langle A, v \rangle_x \mid \Gamma \xrightarrow{A:x,B}_\flat \langle A, v \rangle_x \mid A[x \rhd B] \mid \Gamma} \text{[Dep-AuthDonate]}$$

$$\frac{\Gamma = A[x \rhd B] \mid \Gamma' \quad y \text{ fresh}}{\langle A, v \rangle_x \mid \Gamma \xrightarrow{donate(x,B)}_\flat \langle B, v \rangle_y \mid \Gamma'} \text{[Dep-Donate]}$$

$$\frac{\vec{x} = x_1 \cdots x_n \quad j \in 1..n \quad y \text{ fresh (except in destroy authorizations for } \vec{x})}{\left(\|_{i=1}^n \langle A_i, v_i \rangle_{x_i} \right) \mid \Gamma \xrightarrow{A_j:\vec{x},j}_\flat \left(\|_{i=1}^n \langle A_i, v_i \rangle_{x_i} \right) \mid A_j[\vec{x}, j \rhd y] \mid \Gamma} \text{[Dep-AuthDestroy]}$$

$$\frac{\vec{x} = x_1 \cdots x_n \quad \Gamma = \left(\|_{i=1}^n A_i[\vec{x}, i \rhd y] \right) \mid \Gamma'}{\left(\|_{i=1}^n \langle A_i, v_i \rangle_{x_i} \right) \mid \Gamma \xrightarrow{destroy(\vec{x})}_\flat \Gamma'} \text{[Dep-Destroy]}$$

Figure 8. Semantics of Bitcoin deposits.