Strategic Reasoning for BitML Smart Contracts

Luigi Bellomarini, Marco Favorito and Giuseppe Galano

Bank of Italy

Abstract

In this paper, we study the problem of ATL* model checking for Bitcoin smart contracts when formalized in the BITML high-level formal language. Starting from a BITML contract specification, we show how to construct a concurrent game structure (CGS) that captures its semantics. For a significant subset of BITML, we show that, while we adopt imperfect information and perfect recall semantics, we can retain decidability by reducing our problem to model checking on a CGS with only public actions, which is known to be decidable even in this general setting. We validate our framework through practical applications, including the verification of contract liquidity properties and the strategic analysis of key smart contracts.

Keywords

Bitcoin Smart Contracts, Strategic Reasoning, Liquidity Analysis, Multi-Agent Systems

1. Introduction

Distributed Ledger Technologies (DLT) such as Bitcoin [2] and Ethereum [3] enabled the idea of *smart contracts*, i.e. agreements between untrusted parties that can be automatically enforced without a trusted intermediary [4, 5]. The consensus protocol underlying a DLT ensures that, under certain working assumptions, all network participants agree on the public record of state updates (i.e. *transactions*) and that the execution of the smart contracts is compliant with the contract rules. In this way, the state of each contract is uniquely determined by the sequence of its transactions on the public ledger. In many DLT platforms, transactions are sorted in a data structure called *blockchain*: an append-only, timestamped, tamper-resistant, and cryptographically secure chain of transaction blocks. The form of smart contracts varies depending on the DLT platform on which they are built. In Bitcoin, the stack-based and loop-free (hence not Turing-complete) *Script* language [6] allows to specify, for each unspent transaction, the conditions under which they can be spent, e.g., provide a signature by a certain public key. Despite the limited expressiveness of Script, the integration of on-chain and off-chain interactions, usually in the form of cryptographic protocols, makes Bitcoin able to support a variety of interesting smart contracts [7, 8], such as lotteries [9, 10, 11, 12, 13], gambling games [14], micro-payment channels [15, 16, 17], contingent payments [18, 16], and fair multi-party computations [19, 20].

Smart contracts, like traditional programs or protocols, are not immune to security issues. Attackers may exploit vulnerabilities in implementing contracts or publish contracts with hidden vulnerabilities to control their behavior in ways not expected by users. For example, several vulnerabilities in Ethereum smart contracts have caused the theft or the freezing of Ether or other digital assets worth millions of USD at the time: the famous attacks on The DAO [21], the Parity Multisig Wallet [22, 23, 24], the King of the Ether Throne smart contract [25] are only few notorious examples. Another important source of problems comes from coding errors or design issues that affect the behaviour of the contract at the interaction level: the participants might not be incentivized to behave honestly because they can exploit an unfair advantage with dishonest behaviour [26, 27, 28, 29]. For example, a naive design of a rock-paper-scissors game [30] allows playing sequentially rather than concurrently and gives an

AI4CC-IPS-RCRA-SPIRIT 2024: International Workshop on Artificial Intelligence for Climate Change, Italian Workshop on Planning and Scheduling, RCRA Workshop on Experimental evaluation of algorithms for solving problems with combinatorial explosion, and SPIRIT Workshop on Strategies, Prediction, Interaction, and Reasoning in Italy. November 25-28th, 2024, Bolzano, Italy [1].
All views are those of the authors and do not necessarily reflect the position of Bank of Italy.
luigi.bellomarini@bancaditalia.it (L. Bellomarini); marco.favorito@bancaditalia.it (M. Favorito); giuseppe.galano2@bancaditalia.it (G. Galano)
0000-0001-6863-0162 (L. Bellomarini); 0000-0001-9566-3576 (M. Favorito); 0009-0008-3251-6606 (G. Galano)


advantage to the second player who can see the opponent's move. The DAO attack is an example where a coding bug incentivized dishonest behavior. See [31, 32, 33, 34, 35, 36, 37] for surveys on the matter.

To overcome the mentioned security issues, the industry and the research community have put a lot of effort into developing techniques for *formally specifying and verifying* smart contract correctness. A plethora of different model formalisms, specification languages, and verification techniques studied for traditional programs have been applied or adapted in the context of smart contracts [38, 32, 39, 40, 41]. Regarding formal verification, one of the main used technique is *model checking*, a well-established formal method that automatically checks for system correctness [42, 43, 44, 45]. In this framework, we represent the system into a formal model, typically a state-transition graph such as *Kripke structures* [46] or *labelled transition systems* (LTS) [47], specify the property using temporal logics such as LTL [48], CTL [42], or CTL* [49], and check formally that the model satisfies the temporal logic specification.

However, in model checking, the system is assumed to be *closed*, i.e., a system whose behavior is completely determined by its state. In contrast, an *open* system interacts with multiple participants, and its behaviour depends on this interaction [50, 51]. In fact, most techniques for formally verifying smart contracts treat them as closed systems, disregarding the interaction aspect [52], but a smart contract that does not accept inputs from its participants is not realistic nor interesting. A common workaround is to fix the users' behavior in advance to make the system closed, enabling the use of model checking. While this approach is enough for the verification of interesting properties, e.g. the *(strategyless) liquidity* property [53, 54], it has the major downside of potentially missing adversary strategies that could exploit security vulnerabilities. Adopting *module checking* [51, 55] as a verification framework for open systems would help, but at the cost of only considering two-player games between the participants' coalition and the *environment*, hence limiting the expressiveness of the model and what can be reasoned about. For example, we cannot reason about *strategic abilities* of smart contract users, e.g. which outcome can be enforced or prevented by a single participant or a coalition of participants.

In this paper, we put forward the idea of modeling the interaction between a smart contract and its participants as a *game between agents in a multi-agent system* [56, 57]. In particular, starting from a Bitcoin smart contract, formalized in the high-level modelling language BITML [58], we define a *BITML game* as a *concurrent game structure with imperfect information* (iCGS) [59] that follows the BITML semantics, in the sense that there is a tight correspondence between the runs in the two formalisms. Then, for a relevant subset of BITML contracts, we study the problem of model checking *alternating-time temporal logic* (ATL*) [59] specifications over BITML games. ATL* is a logic formalism that allows reasoning about agents' strategies within a multi-agent system, by introducing path quantifiers to range over those paths that a team of agents can force the system into. Here, we adopt the setting of *imperfect information*, which is needed to model secret commitments, and *perfect recall*. While in general this setting leads to undecidability of the ATL* model checking problem [60], we show how our game structure from BITML contracts can be cast into a *iCGS with only public action* [61], for which our problem remains decidable. Finally, we show that several strategic properties of interest can be captured by ATL* specifications on BITML games, e.g. liquidity [54].

The rest of the paper is structured as follows: Section 2 reviews previous works related to ours; Section 3 introduces the BITML formalism; Section 4 describes our translation from BITML specification to a CGS; Section 5 defines our model checking problem and proves decidability; Section 6 discusses use cases, such as general patterns of liquidity analysis and ATL* model checking with concrete BITML smart contracts; and Section 7 concludes the paper and discusses future works.

2. Related Works

One of the first works that used some form of strategic reasoning in the context of smart contracts was presented in [26], where the authors combined probabilistic model checking and game theory to analyze a specific protocol that also involve players (human users) with different/competing gaming strategies. They used game theory to analyze the protocol's players' behaviour, and then the resulting strategies were modelled in a probabilistic system for automated validation. TypeCoin [62] is a high-

level language for Bitcoin smart contracts that allows the modeling of the updates of a state machine as affine logic propositions. Users can "run" this machine by putting transactions on the blockchain, with the guarantee that only the legit updates can be performed. A downside is that liveness is guaranteed only by assuming cooperative participants, i.e. a dishonest participant can make the others unable to complete an execution, hence they study only a specific type of user strategies. Chatterjee et al. [27] defined a simplified language for loop-free smart contracts, with a compiler to Solidity. Then, they provided an automatic translation into concurrent game structures and analyzed these games employing interval abstraction demonstrated to be sound. The technique is very powerful and of practical relevance, considering that models of concurrent games are very large. The main difference is that they focus on two-player games with quantitative objectives, while ATL* can model more general temporal properties, including some quantitative property (e.g. constraints on deposits), nested games, and reason about strategic abilities of coalitions. Moreover, their framework is suited for Ethereum-like platforms, but not easily adaptable for Bitcoin smart contracts. Tsankov et al. [53] study a property called *liquidity*, which holds when the contract always admits a trace where its balance is decreased (so, the funds stored within the contract do not remain frozen). A major limitation is that their approach considers only cooperating adversaries, and do not handle more complex strategic reasoning. Bartoletti and Zunino [54] study the liquidity property for BITML smart contracts in an adversarial setting, including the verification of general LTL properties. To do so, their tool [63] relies on the Maude model checker [64, 65]. Their approach allows one checking whether the smart contract satisfies the liquidity property regardless of the participants' behaviors or if certain user strategies are fixed beforehand. While being a very useful tool, it is not able to handle branching time properties, coalitions of agents, or complex strategic interactions as one could do in ATL*. The behavior of contract users can also be partially defined in a Turing-complete process calculus `scl` proposed by Laneve et al. [28]. `scl` models behaviors of a smart contract and a user as a parallel composition, where a user acts nondeterministically, while the behavior of a smart contract is determined by its execution logic. The `scl` models are, in fact, transition systems that are summarized by a first-order logic formula describing the values of the objective function for all possible runs and user strategies. Then, the game-theoretical analysis is performed to identify strategies that help users maximize their profits or minimize losses. There are a few differences with our work: *(i)* we use a sequential model rather than process algebra but still model the interleaving between the users; *(ii)* their work considers only rational strategies, while we also aim to find the outcomes of strategies in the general cases of non-cooperative or dishonest (possibly non-rational) behavior; and *(iii)* they only consider quantitative objectives, while we also study the case with temporal qualitative objectives. van der Meyden [29] investigated the specification and verification of an atomic swap smart contract and argued that logics with the ability to express properties of players' strategies in a multi-agent setting, such as ATL, are conceptually useful for this purpose. The MCK model checker was used to establish the correctness and fairness of a contract under the specified user behaviors. However, this approach suffers from the issue of having to fix the user strategy to verify certain properties without letting the system automatically figure out the possible strategies. Madl et al. [66] capture the interaction between the users of a marketplace contract by describing their behaviors as interface automata. The authors verify the composition of the automata in a model checker to ensure that participants can successfully cooperate, but once again, the strategies of the participants are fixed in advance. Nam and Kil [67] propose a technique to translate an Ethereum smart contract written in a subset of the Solidity language into an input file for the MCMAS tool [68] and then use the tool to verify certain ATL specifications. However, their framework does not handle imperfect information, and only focus on two-player games.

3. Overview of BITML

In this section we briefly overview BITML [58, 69]. We assume a set of *participants* $\mathcal{P}$, ranged over by A, B, ..., and a set of names of two kinds: $x, y, \ldots$ denote *deposits* $\mathcal{D}$ of ₿, while $a, b, \ldots$ denote *secrets* $\mathcal{S}$. We also use v, v', w to range over non-negative rational values. We write $\vec{x}$ (resp. $\vec{a}$) for a finite sequence of deposit (resp. secrets) names. We denote with $\mathcal{S}_{\mathsf{A}} \subseteq \mathcal{S}$ the set of secret names usable by A,

requiring that $\mathcal{S}_{\mathsf{A}} \cap \mathcal{S}_{\mathsf{B}} = \emptyset$ if $\mathsf{A} \neq \mathsf{B}$. To facilitate strategic reasoning, we restrict the original semantics by removing contract advertisement rules, deposit rules, and authorizations of deposit operations.

Syntax. BitML is a domain-specific language and a calculus for Bitcoin smart contracts, which allows participants to exchange cryptocurrency according to pre-agreed contract rules. A BitML contract specification, denoted with $\{G\}C$, is made of two components: a *contract* C, that specifies the rules to transfer bitcoins (฿), and *contract preconditions* G, that is a set of preconditions to its execution. In the following, we will use the formalization proposed in [54].

Contract preconditions have the following syntax (the deposits x in a contract precondition G must be distinct): $G ::= \mathsf{A} : ! \; v \; @ x \mid \mathsf{A} : ? \; v @ x \mid \mathsf{A} : \texttt{secret} \; a$. A contract precondition G may require participants to deposit some ฿ in the contract (either upfront or at runtime) or to commit to some secret. More in detail, $\mathsf{A} : ! \; v @ x$ requires A to own v฿ in a *persistent deposit* (or $!$*-deposits*) x, and to spend it for stipulating a contract C. Instead, $\mathsf{A} : ? \; v @ x$ only requires A to pre-authorize the spending of the *volatile deposit* (or $?$*-deposits*) x, which can be gathered by the contract at run-time. The precondition $\mathsf{A} : \texttt{secret} \; a$ requires A to commit to a *secret* a before C starts. After stipulation, A can choose whether to disclose the secret a, assuming the published commitment was not fake.

Contracts are terms with the syntax in Figure 1, where: *(i)* the sum $\sum_{i \in I} D_i$ is over a finite set of indices I; *(ii)* the names $\vec{a}$ in $\texttt{put} \; \vec{x} \; \& \; \texttt{reveal} \; \vec{a} \; \texttt{if} \; p$ are distinct, and they include those in p. We denote the empty sum with 0. The order of decorations is immaterial, e.g., $\texttt{after}\, t : \mathsf{A} : \mathsf{B} : D$ is equivalent to $\mathsf{B} : \mathsf{A} : \texttt{after}\, t : D$. Intuitively, a contract C is a *choice* among zero or more branches. Each branch is a *guarded contract*, which enables an action and possibly proceeds with a continuation C'. The guarded contract $\texttt{withdraw} \; \mathsf{A}$ transfers the whole balance to A, while $\texttt{split} \parallel_i (w_i \to C_i)$ decomposes the contract into n parallel components C_i, each with balance w_i. The guarded contract $\texttt{put} \; \vec{x} \; \& \; \texttt{reveal} \; \vec{a} \; \texttt{if} \; p$ atomically performs the following: *(i)* spend all the $?$-deposits $\vec{x}$, adding their values to the contract balance; *(ii)* check that all the secrets $\vec{a}$ have been revealed and satisfy the predicate p. When enabled, anyone can fire the above-mentioned actions at any time. To restrict *who* can execute actions and when, one can use the decoration $\mathsf{A} : D$, which requires the authorization of A, and the decoration $\texttt{after}\, t : D$, which requires waiting until time t. We use the syntactic sugar $\texttt{reveal} \; a$ for $\texttt{put} \; [] \; \& \; \texttt{reveal} \; a \; \texttt{if} \; \mathit{true}$.

$C ::= \sum_{i \in I} D_i$	contract		$p ::=$		predicate
			true		truth
$D ::=$	guarded contract		$\mid p \wedge p$		conjunction
$\mid \texttt{withdraw} \; \mathsf{A}$	transfer balance to A		$\mid \neg p$		negation
$\mid \texttt{split} \parallel_i (w_i \to C_i)$	split the balance		$\mid E \circ E$	$(\circ \in \{=, <\})$	
$\mid \mathsf{A} : D$	wait for A's authorization		$E ::=$		expression
$\mid \texttt{after}\, t : D$	wait until time t		N		constant
$\mid \texttt{put} \; \vec{x} \; \& \; \texttt{reveal} \; \vec{a} \; \texttt{if} \; p . C$	collect deposits/secrets		$\mid a$		secret
			$\mid E \circ E$	$(\circ \in \{+, -\})$	

Figure 1: Syntax of BitML contracts and preconditions.

A *BitML contract specification* (or *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 $\{G\}C$. Intuitively, $\{G\}C$ is the advertisement of a contract C with preconditions 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. When all the preconditions G have been satisfied, the contract C becomes *stipulated*. The contract starts its execution with a balance, initially set to the sum of the $!$-deposits required by its preconditions. Running C will affect this balance when participants deposit/withdraw funds to/from the contract.

Semantics. We now define the semantics of BitML. A *configuration* Γ is a term with the syntax:

$$\Gamma ::= 0 \mid \{G\}C \mid \langle C, v \rangle_x \mid \langle \mathsf{A}, v \rangle_x \mid \mathsf{A}[\xi] \mid \{\mathsf{A} : a \# N\} \mid \mathsf{A} : a \# N \mid (\Gamma \mid \Gamma') \mid (\Gamma \mid t)$$
$$\xi ::= \# \triangleright \{G\}C \mid x \triangleright \{G\}C \mid x \triangleright D$$

where: *(i)* in a committed secret, $N \in \mathbb{N} \cup \{\bot\}$ (where $\bot$ denotes an ill-formed commitment); *(ii)* in a revealed secret, $N \in \mathbb{N}$; *(iii)* in a configuration, there are no duplicate authorizations; *(iv)* in a configuration containing $\langle \cdots \rangle_x$ and $\langle \cdots \rangle_y$, it must be $x \neq y$; *(v)* there exists at most one term t. We assume that $(|, 0)$ is a commutative monoid, and we denote indexed parallel compositions with $\parallel_i$. We say that Γ is a *timed configuration* when it contains a term t. With $\mathrm{cn}(\Gamma)$, we denote the set of contract

names x such that Γ contains $\langle C, v\rangle_x$, for some C and v, and with $\mathrm{dn}(\Gamma)$ the deposit names. The intuition behind the various terms in configurations is the following: $\langle C, v\rangle_x$ is a stipulated contract storing vɃ, uniquely identified by the name x; $\langle \mathsf{A}, v\rangle_x$ is a deposit of vɃ owned by A, and uniquely identified by the name x; $\mathsf{A}[\xi]$ is A's *authorizations* to perform some action ξ; $\{\mathsf{A} : a\#N\}$ represents A's commitment to a secret N (with $N \in \mathbb{N} \cup \{\bot\}$), identified by a; $\mathsf{A} : a\#N$ represents a secret N (with $N \in \mathbb{N}$), identified by a, and revealed by A. Regarding authorizations: $\# \triangleright \{G\} C$ authorizes to commit secrets to stipulate $\{G\} C$; $x \triangleright \{G\} C$ authorizes to spend the !-deposit x to stipulate $\{G\} C$; and $x \triangleright D$ is an authorization to take branch D.

The semantics of a BitML contract specification is a *labelled transition system* (LTS) between timed configurations. A LTS is a triple $\mathcal{M} = (S, \Lambda, T)$ where S is a set of *states*, Λ is a set of *labels*, and $T : S \times \Lambda \to S$ is the *(deterministic) labelled transition function*. We say there is a transition from state p to state q with label ℓ iff $T(p, \ell) = q$. A *run* $\mathcal{R}$ is a (possibly infinite) sequence $\Gamma_0 \mid t_0 \xrightarrow{\ell_0} \Gamma_1 \mid t_1 \xrightarrow{\ell_1} \cdots$ where ℓ_i are the transition labels, Γ_0 is an initial configuration, and $t_0 = 0$. If $\mathcal{R}$ is finite, we write $\Gamma_{\mathcal{R}}$ for its last untimed configuration, and $\delta_{\mathcal{R}}$ for its last time. We write $\mathcal{R} \xrightarrow{\ell} \mathcal{R}'$ when $\mathcal{R}'$ extends $\mathcal{R}$ with the transition $\Gamma_{\mathcal{R}} \mid \delta_{\mathcal{R}} \xrightarrow{\ell} \Gamma_{\mathcal{R}'} \mid \delta_{\mathcal{R}'}$. We write $\mathcal{R} \to^* \mathcal{R}'$ to say that there exists a sequence of transitions such that $\mathcal{R}'$ can extend $\mathcal{R}$.

Below, we introduce the BitML semantics, formally defined in Figure 2. Labels represent the actions performed by participants. A decoration $\mathsf{A} : \cdots$ in the label means that the action can be performed only by A, while its absence means that it can be performed by anyone. Note that labels are not instrumental to defining the BitML semantics. Yet, they are essential in the game-theoretic formalization of a BitML contract since we need to associate actions with the participants who can perform them.

Secret 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. These steps are formalized by rule [C-AuthCommit], where the term $\{\mathsf{A} : a\#N\}$ represents A's commitment to the secret N, while $\mathsf{A}[\# \triangleright \{G\} C]$ represents finalising the commitment phase for A. The rule preconditions ensures that the final configuration fulfills the conditions required by G. Note that condition *(iii)* in the definition of configurations ensures that rule [C-AuthCommit] cannot be used more than once to generate the same authorization. The same is true for all the other rules that generate authorizations.

$$\frac{\begin{array}{ll} a_1 \cdots a_k \text{ secrets of } \mathsf{A} \text{ in } G & \forall i \in 1..k : \nexists N : \{\mathsf{A} : a_i\#N\} \in \Gamma \\ \Delta = \|_{i=1}^{k} \{\mathsf{A} : a_i\#N_i\} & \forall i \in 1..k : \nexists N : \mathsf{A} : a_i\#N \text{ in } \Gamma \\ & \forall i \in 1..k : N_i \in \mathbb{N} \cup \{\bot\} \end{array}}{\Gamma \xrightarrow{\mathsf{A}:\Delta} \Gamma \mid \Delta \mid \mathsf{A}[\# \triangleright \{G\} C]} \text{[C-AuthCommit]}$$

$$\frac{\begin{array}{ll} G = & (\,\|_{i\in I}\, \mathsf{A}_i : !\, v_i \,@\, x_i) \mid (\,\|_{i\in J}\, \mathsf{B}_i : ?\, v'_i \,@\, y_i) \mid (\,\|_{i\in K}\, \mathsf{C}_i : \mathtt{secret}\, a_i) \quad x \text{ fresh} \\ \Delta = & (\,\|_{i\in I}\, \langle \mathsf{A}_i, v_i\rangle_{x_i}) \mid (\,\|_{i\in I}\, \mathsf{A}_i[x_i \triangleright \{G\} C]) \mid (\,\|_{i\in K}\, \mathsf{C}_i[\# \triangleright \{G\} C]) \end{array}}{\Delta \mid \Gamma \xrightarrow{init} \langle C, \textstyle\sum_{i\in I} v_i\rangle_x \mid \Gamma} \text{[C-Init]}$$

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

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

$$\frac{N \neq \bot}{\{\mathsf{A} : a\#N\} \mid \Gamma \xrightarrow{\mathsf{A}:a} \mathsf{A} : a\#N \mid \Gamma} \text{[C-AuthRev]}$$

$$\frac{\begin{array}{lll} \vec{x} = x_1 \cdots x_m & \Gamma = \|_{i=1}^{m} \langle \mathsf{A}_i, v_i\rangle_{x_i} & z \text{ fresh} \\ \vec{a} = a_1 \cdots a_n & \Delta = \|_{i=1}^{n} \mathsf{B}_i : a_i\#N_i & [\![p]\!]_\Delta = true \end{array}}{\langle \mathtt{put}\ \vec{x}\ \&\ \mathtt{reveal}\ \vec{a}\ \mathtt{if}\ p.C, v\rangle_y \mid \Gamma \mid \Delta \mid \Gamma' \xrightarrow{put(\vec{x},\vec{a},y)} \langle C, v + \textstyle\sum_{i=1}^{m} v_i\rangle_z \mid \Gamma' \mid \Delta} \text{[C-PutRev]}$$

$$\begin{array}{c} [\![true]\!]_\Delta = true \quad [\![p_1 \wedge p_2]\!]_\Delta = [\![p_1]\!]_\Delta \text{ and } [\![p_2]\!]_\Delta \quad [\![\neg p]\!]_\Delta = \text{not } [\![p]\!]_\Delta \\ [\![a]\!]_\Delta = N \quad \text{if } \Gamma \text{ contains } \mathsf{A} : a\#N \quad [\![E_1 \bullet E_2]\!]_\Delta = [\![E_1]\!]_\Delta \bullet [\![E_2]\!]_\Delta \quad (\bullet \in \{+, -\}) \\ [\![N]\!]_\Delta = N \quad [\![E_1 \circ E_2]\!]_\Delta = [\![E_1]\!]_\Delta \circ [\![E_2]\!]_\Delta \quad (\circ \in \{=, <\}) \end{array}$$

$$\frac{D \equiv \mathsf{A} : D'}{\langle \mathsf{A} : D + C, v\rangle_x \mid \Gamma \xrightarrow{\mathsf{A}:(x,\mathsf{A}:D)} \langle \mathsf{A} : D + C, v\rangle_x \mid \mathsf{A}[x \triangleright \mathsf{A} : D] \mid \Gamma} \text{[C-AuthBranch]}$$

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

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

Figure 2: Inference rules for the BitML semantics.

Initialization. Once all the needed authorizations have been granted, the advertisement can be turned into an active contract. Rule [C-Init] 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. Note that the part Δ of the configuration contains all the terms that are consumed by the step. The next rules define the behaviour of a contract after stipulation.

Withdraw. Executing `withdraw` A terminates the contract and transfer its balance to A. After the contract x is terminated, a fresh deposit of v₿ owned by A is created. This is defined by rule [C-Withdraw]. The case where the action `withdraw` A has an alternative branch is handled by rule [C-Branch] below.

Split. The `split` primitive divides the contract balance in parts, each one controlled by its own contract. The general rule is [C-Split]. Note that the weights w_i in the split do not represent actual ₿ values, but the proportion w.r.t. the contract balance.

Revealing secrets. Any participant can reveal one of her secrets, using the rule [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 $\mathsf{A} : a$ represents the fact that only A, the participant who performed the commitment, can fire the transition.

Put-Reveal. The prefix `put` $\vec{x}$ & `reveal` $\vec{a}$ `if` p can be fired with rule [C-PutRev] if all the deposits $\vec{x}$ are available, all the committed secrets $\vec{a}$ have been revealed, and satisfy the guard p, where $[\![p]\!]_\Delta$ is the evaluation of predicate p in configuration Δ.

Authorizing branches. With rule [C-AuthBranch], a branch $\mathsf{A} : D$ can be taken only provided that A has granted her authorization.

Reducing branches. Once all the authorizations for a branch occur in the configuration, anyone can trigger the transition with rule [C-Branch], provided that the time constraints (if any) are respected.

Delaying. In any configuration, we allow time to advance with rule [C-Delay].

In this paper, we consider the LTS $\mathcal{M}_{\{G\}C}$, associated with a contract specification $\{G\}C$, with only the participants occurring in G and as initial timed configuration $\Gamma_0 \mid t_0$ (with $t_0 = 0$) the set of persistent deposits x_i and volatile deposits y_j occurring in G held by their respective participants, as well as the authorizations $\mathsf{A}_k[x_i \triangleright \{G\}C]$ to spend the persistent deposits. While the original semantics of BitML included the possibility of fresh contract advertisement and manipulation of deposits, here we strip out it by leaving only the needed elements to reason on a single BitML contract. These simplifications rule out some interesting participants' behaviors that were possible in the original semantics, e.g., advertising a new contract or destroying the deposits to prevent specific contract branch executions. Due to space limitations, we preferred to focus on the core features of a BitML contract and the main strategic reasoning tasks we can perform.

4. BitML Games

In this section, we are interested in defining a game structure that models the interaction between participants of a BitML smart contract. Given a set X of elements, let $u \in X^\omega \cup X^+$ denote an infinite or non-empty finite sequence on X. Then, we write u_i for its $(i+1)$-th element, i.e., $u = u_0 u_1 \ldots$, and $|u|$ for its length, with $|u| = \infty$ for $u \in X^\omega$. The first element of u is denoted by $\mathit{first}(u)$, and its last by $\mathit{last}(u)$. We write $u_{\geq i}$ for its suffix $u_i u_{i+1} \ldots$ starting in u_i, and $u_{\leq i}$ for its prefix $u_0 \ldots u_i$. The empty sequence is denoted by ϵ. For a vector $v \in \Pi_i X_i$, we denote its i-th element by $v(i)$.

Games and strategies. A *concurrent game structure with imperfect information* (iCGS) [59] is a tuple $\mathcal{G} = \langle \mathrm{St}, \mathrm{AP}, \mathrm{Ag}, \{\mathrm{Act}_a\}_{a\in\mathrm{Ag}}, \lambda, \tau, S_\iota, \{\sim_a\}_{a\in\mathrm{Ag}}\rangle$ where: *(i)* St is a finite non-empty set of *states*; *(ii)* AP is a finite non-empty set of *atomic propositions*; *(iii)* Ag is a finite non-empty set of *agents* ($m = |\mathrm{Ag}|$); Act_a is a set of *actions* of agent $a \in \mathrm{Ag}$, and with every state s and agent a we associate a subset $\mathrm{Act}_a(s) \subseteq \mathrm{Act}_a$ of actions that a can perform at s; *(iv)* $S_\iota \in \mathrm{St}$ is the *set of initial states*; *(iv)* $\lambda : \mathrm{St} \to 2^{\mathrm{AP}}$ is a *labelling function*; *(v)* For each state $s \in \mathrm{St}$ and each *joint action* $J \in \mathrm{Jact}(s) = \prod_{a\in\mathrm{Ag}} \mathrm{Act}_a(s)$, we define the state $\tau(s, J) \in \mathrm{St}$ that results from state s if every player

a chooses move j_a, with τ called *transition function*; *(vi)* for each agent $a \in \mathrm{Ag}$, $\sim_a \subseteq \mathrm{St} \times \mathrm{St}$ is an equivalence relation called *indistinguishability relation*.

Intuitively, the game starts in an initial state $s_\iota \in S_\iota$, and in any state s each agent a chooses an action $j_a \in \mathrm{Act}_a(s)$, and the game proceeds to state $s' = \tau(s, J)$, where $J \in \mathrm{Jact}(s)$ stands for the joint action allowed in state s, and $J(a) = j_a$ is the action of agent a in J. For each state $s \in \mathrm{St}$, $\lambda(s)$ is the finite set of atomic propositions that hold in s, and for $a \in \mathrm{Ag}$, $\sim_a$ is an equivalence relation that represents the observation of agent a: for two states $s, s' \in \mathrm{St}$, $s \sim_a s'$ means that agent a cannot tell the difference between s and s'. We say that s' is a *successor* of s, with $s, s' \in \mathrm{St}$, if there exists a joint action J such that $\tau(s, J) = s'$. A *run* $\rho \in \mathrm{St}^\omega$ is an infinite sequence $\rho = s_0 s_1 s_2 \cdots \in \mathrm{St}^\omega$ such that $s_0 \in S_\iota$ and for every integer $i \geq 0$, s_{i+1} is a successor of s_i. The set of runs is denoted by Runs. A *history* $h \in \mathrm{St}^+$ is a finite sequence $h = s_0 s_1 s_2 \dots s_k$ such that $s_0 \in S_\iota$ and for $0 \leq i < k$, s_{i+1} is a successor of s_0. The set of histories is denoted by Hist. Each indistinguishability relation is thus extended to runs and as follows: $\rho \approx_a \rho'$ if $\rho_i \sim_a \rho'_i$ for every $i \geq 0$. Similarly, for $h, h' \in \mathrm{Hist}$, $h \approx_a h'$ if $|h| = |h'|$ and $\rho_i \sim_a \rho'_i$ for every $0 \leq i < |h|$. A *fairness constraint* is a criterion to distinguish between fair and unfair runs. Here, we use *unconditional* fairness constraints, defined as a subset of states $fc \in 2^{\mathrm{St}}$, as in [70]. A run ρ is *fair* according to a set of fairness conditions $\mathrm{FC} = \{fc_1, \dots, fc_k\}$ if for each fairness condition fc_j, there exist infinitely many positions i such that $\rho(i) \in fc_j$.

A (*perfect recall or memoryful*) *strategy* for agent a in a iCGS $\mathcal{G}$ is a function $\sigma_a : \mathrm{Hist} \to \mathrm{Act}_a$ such that for all histories h, $\sigma_a(last(h)) \in \mathrm{Act}_a(last(h))$, and for all histories h, h', $\sigma_a(h) = \sigma_a(h')$ whenever $h \approx_a h'$. The latter constraint captures the essence of imperfect information: agents can base their strategic choices only on the information available to them. For $A \subseteq \mathrm{Ag}$ and $y \in \{R, r\}$, let $\sigma_A : A \to \Sigma_y(G)$ denote a joint strategy associating a memoryful strategy σ_a with each agent $a \in A$. For $h \in \mathrm{Hist}$ and a joint strategy σ_A, we write $\mathrm{Out}(h, \sigma_A)$, called the *outcomes of* σ_A *from* h, for the set of runs $\rho \in \mathrm{Runs}$ such that ρ is consistent with h and σ_A. That is, $\rho \in \mathrm{Out}(h, \sigma_A)$ iff *(i)* h is a prefix of ρ; *(ii)* for every $i \geq |h| - 1$, there exists a joint action $J_i \in \mathrm{Jact}$ such that $\rho_{i+1} = \tau(\rho_i, J_i)$ and for every $a \in A$, $J_i(a) = \sigma_A(a)(\rho_{\leq i})$. Moreover, we define $\mathrm{Out}^i(h, \sigma_A) = \bigcup_{h \approx_A h'} \mathrm{Out}(h', \sigma_A)$, where $\sim_A = \bigcup_{a \in A} \sim_a$ and $\approx_A = \bigcup_{a \in A} \approx_a$, which represents the set of outcomes under subjective semantics [71] and the "everybody knows" type of collective knowledge [72].

From BitML Contracts to Game Structures. To enable strategic reasoning on BitML smart contracts, in this section, we show how to compute a iCGS $\mathcal{G}_{\{G\}C}$, starting from a BitML contract specification $\{G\}C$. The game structure $\mathcal{G}_{\{G\}C}$ will be designed so that the plays generated in $\mathcal{G}_{\{G\}C}$ have a corresponding run in the LTS $\mathcal{M}_{\{G\}C}$, and vice-versa. This will enable the application of multi-agent strategic reasoning techniques based on the formal model of game structures, which resembles the BitML semantics. Note that because agents can commit to arbitrary secret strings and can advance the current time (a natural number) for an arbitrary time delay [54], $\mathcal{M}_{\{G\}C}$ has an infinite state space and infinite label space, and so the game structure $\mathcal{G}_{\{G\}C}$, hence hindering the feasibility of reasoning techniques. In the next section, we show how to abstract the secret commitments and the time representation to make the state and action spaces finite.

Before presenting the formal definition, we explain our design choices for $\mathcal{G}_{\{G\}C}$. (*a*) Firstly, we have to depart from a concurrent formalization and instead focus on a turn-based game since the inference rules of the BitML semantics (except the [C-Delay] rule) are associated with actions to be performed by a single agent. This is also reflected in the instantaneous nature of these actions, similar to many timed process calculi [73]. Hence, we can rule out agent actions that happen simultaneously with others. (*b*) Secondly, the interaction between the agents must be asynchronous, meaning that, from the perspective of a single agent, there is some nondeterminism that determines whose turn it is. We can rely on a *turn-based asynchronous game structure* by introducing a fair scheduler agent that decides which agent plays at each step (e.g. see [59, 74]), and all the other agents are forced to play the *null action* ε. Multi-agent systems with asynchronous execution, often formalized as *interleaved interpreted systems* (IIS) [72], are a well-known setting in the verification of multi-agent systems [44, 59, 75, 76, 77]. (*c*) Thirdly, some mechanisms in the game structure must guarantee that the adversary cannot prevent the execution of the contract unless the contract itself permits this. More precisely, we want to rule out the

cases in which, during the initialization phase, some agents never commit to secrets via [C-AuthCommit], nor anyone initializes the contract via [C-Init]. This is required to focus on the analysis of the contract rather than the participants' decision to participate in it. (*d*) Finally, we require that the progress of time is agreed upon by all participants, and that there is a fairness constraint that guarantees that eventually the time moves forward. This is crucial to comply with the assumption that participants can always meet deadlines if they want to.

Thus, we define the game structure $\mathcal{G}_{\{G\}C}$ as follows:
• St is the set of tuples $\langle \Gamma, t, Aux \rangle$, i.e. the set of all timed configurations reachable from $\Gamma_0 \mid t_0$ in $\mathcal{M}_{\{G\}C}$, plus a set of auxiliary predicates Aux for defining the fairness constraints. We define $\mathrm{Conf}(s) = \Gamma$, $\mathrm{Time}(s) = t$, and $\mathrm{Aux}(s) = Aux$. The (unique) initial state is $s_\iota = \langle \Gamma_0, t_0, \emptyset \rangle$.
• $\mathrm{Ag} = \mathcal{P} \cup \{\mathsf{Sch}\}$, where $\mathcal{P}$ is the set of participants and Sch is the *scheduler* agent;
• Act_a: if $a \in \mathcal{P}$, then for each rule of the BitML semantics, except [C-Delay], we have a corresponding action, e.g.: action "$\mathsf{A} : \{\mathsf{A} : a_1 \# N_1\}, \ldots, \{\mathsf{A} : a_k \# N_k\}$" from [C-AuthCommit], $init$ from [C-Init]; $withdraw(\mathsf{A}, v, x)$ from [C-Withdraw], and so on. To allow agents to progress the time synchronously, we include a new action, $done_{\mathsf{A}}$, meaning that agent A agrees to progress the time by one unit. The effect of this action is to set the auxiliary proposition $\mathsf{Done}_{\mathsf{A}}$ to true in the next state. If $\mathsf{Done}_{\mathsf{A}} \in \mathrm{Aux}(s)$, we say that participant A *ready to progress the time* (or *delay the time*) in state s, and we denote with the term $\mathrm{Ready}(s)$ the set of agents that have ready to delay the time in s. Each agent also has the action ε, with no effect on the current state. If instead $a = \mathsf{Sch}$, we have an action for each agent in $\mathcal{P}$ plus the action $delay$ that increases the current time by one unit. In a state s, if $\mathrm{Ready}(s) = \mathcal{P}$, then the only enabled action is $delay$; otherwise, the set of available actions is $\mathcal{P} \setminus \mathrm{Ready}(s)$, i.e., the scheduler can assign the turn to an agent that is not ready yet to progress the time.
• The set of proposition AP captures relevant features of the game states. The actual choice for AP and λ will depend on the specification to be verified. Some examples will be discussed in the next sections.
• Being $\mathcal{G}_{\{G\}C}$ a turn-based asynchronous game, the transition function τ is defined according to the construction in [59]. In particular, for all states $s \in \mathrm{St}$ and all joint actions $J_1, J_2 \in \mathrm{Jact}(s)$, if $J_1(\mathsf{Sch}) = J_2(\mathsf{Sch}) = a$ and $J_1(a) = J_2(a)$, then $\tau(s, J_1) = \tau(s, J_2)$. The effects of the auxiliary actions have been specified above, and the ones from the BitML semantics are specified by the inference rules.
• The indistinguishability relations $\sim_a$, one for each participant $a \in \mathcal{P}$, captures partial observability on the value of the secrets of the other participants $\mathcal{P} \setminus \{a\}$. In other words, $\sim_a$ is such that states where other agents have committed to different secrets are all indistinguishable from a's point of view.
• We add further auxiliary propositions in order to define our desired fairness constraints: *(i)* during the stipulation phase, each participant must eventually execute the action corresponding to [C-AuthCommit]; *(ii)* after that, some participants must eventually execute the $init$ action to initialize the contract; *(iii)* the scheduler is fair, i.e., it does not neglect an agent forever; *(iv)* each participant is eventually ready to progress the time. We could define a fairness constraint for each action involved by adding auxiliary propositions in the Aux component of states, e.g., $\mathsf{AuthCommit}_{\mathsf{A}}$ to keep track of whether A has committed to secrets in the past, $\mathsf{Initialized}$ to keep track of whether the contract has been initialized, $\mathsf{Acted}_{\mathsf{A}}$ that holds in the current state if A was the last agent scheduled by Sch, and $\mathsf{Done}_{\mathsf{A}}$ (already defined above) to keep track whether A has taken action $done_{\mathsf{A}}$ since the most recent time delay, and $\mathsf{Delayed}$ if the last action of Sch was $delay$. Given these auxiliary propositions, defining a fairness constraint for each requirement above is straightforward.

We observe that such construction complies with the abovementioned design choices (*a*), (*b*), (*c*) and (*d*). Also, the time progression machinery is not needed if there are no timeouts in the contracts. Moreover, note that we make an implicit assumption, often made in model checking cryptographic protocols, that is called "perfect cryptography". This assumption implies that the probability of adversaries discovering the secret value through brute force is zero [29].

Correspondence between the semantics. We show that each *trajectory*, i.e., sequence of state and actions $S = s_0, J_0, s_1, J_1, \ldots$ compatible with τ in the game structure $\mathcal{G}_{\{G\}C}$, has a corresponding run $\mathcal{R}$ of the LTS $\mathcal{M}_{\{G\}C}$, in the sense that S and $\mathcal{R}$ visit the same configurations by doing analogous transitions (modulo auxiliary propositions and synchronization actions). The correspondence also

holds in the other direction: given a run $\mathcal{R}$, we can construct a trajectory S in $\mathcal{G}_{\{G\}C}$ where the configurations and the actions in $\mathcal{R}$ occur in the same order. We give further definitions to formalize such correspondence before formally proving the result. We inductively define a transformation from trajectories $S = s_0, J_1, s_1, \ldots$ of $\mathcal{G}_{\{G\}C}$ to runs $\mathcal{R}$ of $\mathcal{M}_{\{G\}C}$, denoted with tr: for $S_0 = s_0$, where $s_0 = s_\iota = \langle \Gamma_0, t_0, Aux_0 \rangle$, we have $tr(S_0) = \Gamma_0 \mid t_0$. Let $S_k = s_0, J_1, \ldots, J_k, s_k$ be a generic sequence in $\mathcal{G}_{\{G\}C}$ of length k, and $\mathcal{R} = tr(S_k)$ be its associated run on $\mathcal{M}_{\{G\}C}$. Consider a valid extension of S_k, i.e. $S_{k+1} = S, J_{k+1}, s_{k+1}$ such that $\tau(s_k, J_{k+1}) = s_{k+1}$. In case $J(\mathsf{Sch}) = delay$, then $tr(S_{k+1}) = \mathcal{R} \xrightarrow{1} \Gamma_{k+1} \mid t_{k+1}$, be the extension of $\mathcal{R}$ after applying rule [C-Delay] with delay 1, with $\Gamma_{k+1} = \Gamma_k$ and $t_{k+1} = t_k + 1$. Otherwise, in case $J(\mathsf{Sch}) = \mathsf{A}$ for some $\mathsf{A} \in \mathcal{P}$ that has not yet ready to progress the time in s_k: *(i)* if $J_k(\mathsf{A})$ is one of ε or $done_{\mathsf{A}}$, then $tr(S_{k+1}) = tr(S_k)$; *(ii)* otherwise, $tr(S_{k+1}) = \mathcal{R} \xrightarrow{\ell} \Gamma_{k+1} \mid t_{k+1}$, where ℓ is the label corresponding to $J_{k+1}(\mathsf{A})$ in the LTS $\mathcal{M}_{\{G\}C}$, Γ_{k+1} is the configuration obtained after taking action $J_{k+1}(\mathsf{A})$ in state s_k, and $t_{k+1} = t_k$.

Theorem 1. *Let $\{G\}C$ be a BitML contract specification. Let $S = s_0, J_1, s_1, \ldots$ be a trajectory on $\mathcal{G}_{\{G\}C}$. Then, $\mathcal{R} = tr(S)$ is a run of $\mathcal{M}_{\{G\}C}$.*

Proof sketch. The claim can be proved by the definition of tr, by the construction of $\mathcal{G}_{\{G\}C}$, and by induction on the length of the trajectory. The idea is that auxiliary actions in S, e.g., ε and $done_{\mathsf{A}}$ are trimmed out by tr, while the other actions are all translated in $\mathcal{R}$ into legal moves of $\mathcal{M}_{\{G\}C}$. □

Moreover, each run $\mathcal{R}$ of the LTS $\mathcal{M}_{\{G\}C}$ has a corresponding trajectory S in $\mathcal{G}_{\{G\}C}$:

Theorem 2. *Let $\{G\}C$ be a BitML contract specification. Let $\mathcal{R} = \Gamma_0 \mid t_0 \xrightarrow{\ell_1} \Gamma_1 \mid t_1 \xrightarrow{\ell_2} \cdots \xrightarrow{\ell_j} \Gamma_j \mid t_j$ be a run of $\mathcal{M}_{\{G\}C}$. There exist a trajectory $S = s_0, J_1, s_1, \ldots J_k, s_k$ of $\mathcal{G}_{\{G\}C}$ such that $tr(S) = \mathcal{R}$.*

Proof sketch. The claim can be proved by the definition of tr, by the construction of $\mathcal{G}_{\{G\}C}$, and by induction on the length of the run $\mathcal{R}$. Intuitively, S has the same actions as $\mathcal{R}$, but whenever there is a time delay of δ, all the agents synchronize for δ rounds to increment the current time of δ units. □

5. ATL* model checking on BitML Games

In this section, we propose a unifying framework based on ATL* model checking, in which BitML game structures $\mathcal{G}_{\{G\}C}$ can be verified against ATL* specifications.

Alternating-time Temporal Logic. First, we introduce *Alternating-time Temporal Logic* (ATL*) [59, 78], a well-known and popular logic formalism for modeling and verifying multi-agent systems. The temporal logic ATL* is defined with respect to a finite set of propositions AP and a finite set $\mathrm{Ag} = \{a_1, \ldots, a_m\}$ of players. The syntax of ATL* is given by the grammar

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\!\langle A \rangle\!\rangle \psi \qquad\qquad \psi ::= \varphi \mid \neg\psi \mid \psi \wedge \psi \mid \bigcirc\psi \mid \psi \,\mathcal{U}\, \psi$$

where $p \in \mathrm{AP}$ and $A \subseteq \mathrm{Ag}$. The *temporal operators* are $\bigcirc$ ("next") and $\mathcal{U}$ ("until"). The *strategy quantifier* is $\langle\!\langle A \rangle\!\rangle$, meaning "the agents in A can enforce ψ". The logic ATL* is similar to the branching-time temporal logic CTL* [42], only that path quantifiers are parameterized by sets of players. Sometimes we write $\langle\!\langle a_1, \ldots, a_l \rangle\!\rangle$ instead of $\langle\!\langle \{a_1, \ldots, a_l\} \rangle\!\rangle$, and $\langle\!\langle \rangle\!\rangle$ instead of $\langle\!\langle \emptyset \rangle\!\rangle$. Additional Boolean connectives are defined from $\neg$ and $\wedge$ in the usual manner. Moreover, we define "sometime in the future" as $\Diamond\psi \equiv \top \,\mathcal{U}\, \psi$ and "always in the future" as $\Box\psi \equiv \neg\Diamond\neg\psi$. formulas φ and ψ are called *state* and *path formulas* of ATL*, respectively. State formulas constitute the language of ATL*. By imposing that temporal operators are immediately preceded by a strategic operator, we obtain the logic fragment ATL.

Since Alur et al.'s seminal paper, several different semantics of ATL* have been proposed. For instance, agents may be able to observe the full state of the system or only parts of it (*perfect* vs. *imperfect information*), and they may base their decisions on the current state only, or on the entire history of the game (*perfect* vs. *imperfect recall*) [79, 80, 81]. Moreover, agents can have *objective* or *subjective* ability to achieve their goals [71], their behavior can be subject to fairness constraints [59, 70], or they can be endowed with a mechanism for broadcasting information within a team [82, 83]. Another concern is how collective knowledge of a coalition A is interpreted [72]. The most used variants are: agents with *mutual knowledge* (also known as "everybody knows" [79], where agents aggregate uncertainty), agents

with *distributed knowledge* (where agents aggregate certainty), and agents with *common knowledge* (where agents aggregate uncertainty, plus uncertainty about uncertainty etc.).

In this paper, we assume *(i)* perfect recall, *(ii)* imperfect information with subjective semantics, and *(iii)* mutual knowledge as collective knowledge relation. We use the so-called *truly perfect recall* semantics [81, 61]. We simultaneously define, by induction on the formulas, $(\mathcal{G}, h) \models \varphi$ where $h \in \text{Hist}$ and φ is a history formula, and $(\mathcal{G}, \rho, k) \models \psi$ where $\rho \in \text{Runs}$, $k \geq 0$, and ψ is a path formula:
- $(\mathcal{G}, h) \models p$ iff $p \in \lambda(last(h))$, for $p \in \text{AP}$;
- $(\mathcal{G}, h) \models \langle\!\langle A \rangle\!\rangle \psi$ iff $\exists \sigma_A \in \Sigma_R(\mathcal{G})$ such that, $\forall \rho \in \text{Out}^i(h, \sigma_A)$, we have $(\mathcal{G}, \rho, |h| - 1) \models \psi$
- $(\mathcal{G}, \rho, k) \models \varphi$ iff $(\mathcal{G}, \rho_{\leq k}) \models \varphi$ for φ a state formula;
- $(\mathcal{G}, \rho, k) \models \bigcirc \psi$ iff $(\mathcal{G}, \rho, k+1) \models \psi$;
- $(\mathcal{G}, \rho, k) \models \psi_1 \,\mathcal{U}\, \psi_2$ iff $\exists i.i \geq k$ such that $(\mathcal{G}, \rho, i) \models \varphi_2$ and $\forall j.k \leq j < i$ we have $(\mathcal{G}, \rho, j) \models \varphi_1$.

Boolean operators are omitted. For a state formula φ we say that $\mathcal{G} \models \varphi$ iff $(\mathcal{G}, \langle s_\iota \rangle) \models \varphi$ for all $s_\iota \in S_\iota$.

ATL* Model Checking of BitML Games. In the following, we study the problem of model checking an ATL* specification φ over BitML games. First, observe that $\mathcal{G}_{\{G\}C}$ allows for uninteresting outcomes, such as an unfair scheduler, the time never progresses, or the contract is never initialized. To rule out unfair runs (see requirements *(b)*, *(c)* and *(d)*), we include a set of fairness constraints in φ of the form $\Box\Diamond p$, one for each possible value of p, namely $\mathsf{AuthCommit}_\mathsf{A}$, $\mathsf{Acted}_\mathsf{A}$, Done_A for all $\mathsf{A} \in \mathcal{P}$, and Delayed. Let *Fair* be the conjunction of these constraints, and let $\models_F$ be defined as $\mathcal{G} \models_F \varphi$ iff $\mathcal{G} \models \langle\!\langle \rangle\!\rangle (\mathit{Fair} \rightarrow \varphi)$. Also, we introduce a suitable set of propositions $\text{AP}_{\{G\}C}$, and its associated labeling functions $\lambda_{\{G\}C}$, for $\mathcal{G}_{\{G\}C}$, to provide the alphabet for constructing the ATL* specifications:

- $\mathsf{initialized}_C$: this proposition is false in every initial state, and it is set to true by action *init*. Intuitively, $\mathsf{initialized}_C$ keeps track of whether the contract C has been initialized or not in the current game.
- $\mathsf{committed}_a$ (with $a \in \mathcal{S}$): this type of proposition holds in a state s iff, after contract initialization, there exists a commitment for secret a in $\text{Conf}(s)$: $\{\mathsf{A} : a\#N\} \in \text{Conf}(s)$, for some $N \in \mathbb{N} \cup \bot$.
- $\mathsf{revealed}_a$ (with $a \in \mathcal{S}$): this proposition holds in a state s iff secret a has been revealed in $\text{Conf}(s)$.
- $\mathsf{liquidated}_C$: the proposition $\mathsf{liquidated}_C$ holds whenever there are no contract names, descendant from C, in the current state s, i.e. $\mathsf{liquidated}_C \equiv \mathsf{initialized}_C \wedge (\text{cn}(\text{Conf}(s)) = \emptyset)$. Intuitively, since in BitML active contracts always store some funds, this is equivalent to checking that funds deposited in the contract at initialization time have been withdrawn, so there are no frozen funds. The presence of the $\mathsf{initialized}_C$ is needed to rule out the case in which C has not been initialized yet.
- "$\mathsf{total\text{-}deposits}_\mathsf{A} \circ v$" (where $\circ \in \{=, <\}$): this proposition holds in a state s iff the sum of deposits available to A in $\text{Conf}(s)$ satisfies the condition $\circ$ against the value v, where $\mathsf{total\text{-}deposits}_\mathsf{A} = \sum_{\langle \mathsf{A}, v' \rangle_x \in \text{Conf}(s)} v'$ is the sum of the values of all deposits controlled by A.
- $\mathsf{expired}_t$: this proposition holds in a state s iff $\text{Time}(s) \geq t$, meaning that the timeout t has expired. The term t must occur in some `after`-clause in C.

According to the smart contract and the ATL specification of interest, there might be other choices of atomic propositions. We designed these propositions based on the criterion that they should capture generic properties of contract execution. Finally, we define the problem of *ATL* model checking* a specification φ against a BitML $\mathcal{G}_{\{G\}C}$ as the problem of determining whether $\mathcal{G}_{\{G\}C} \models_F \varphi$.

Game Abstraction. The main limitation in solving our model checking problem is that $\mathcal{G}_{\{G\}C}$ has an infinite state space and an infinite action space. The source of the infiniteness of the state space and the action space comes from *(i)* the commitment of secrets of arbitrary value, and *(ii)* the unboundedness of time. Therefore, naively adopting an off-the-shelf ATL* model checking algorithm (see, e.g., [59]) for solving our problem will not work since the termination is guaranteed only when the state space is finite. In this section, analogous to the abstract semantics of the BitML semantics introduced in [54], we introduce an *abstraction for the (concrete) game* $\mathcal{G}_{\{G\}C}$, denoted with $\mathcal{G}^\sharp_{\{G\}C}$, which reduces the state/action space to a finite one, while guaranteeing correctness of ATL* model checking.

To cope with *(i)*, we make the following simplifying assumption:

Assumption 1. All the clauses `put` $\vec{x}$ `&` `reveal` $\vec{a}$ `if` p in BitML contract C are such that $p = \mathit{true}$.

Assumption 1 simplifies the design of $\mathcal{G}^{\sharp}_{\{G\}C}$ by only considering whether participants have committed to secrets and, if so, whether they are willing to reveal them. In particular, the secret commitment component $\{\mathsf{A} : a\#N\}$, with $N \in \mathbb{N} \cup \{\bot\}$, is replaced with an *abstract secret commitment* $\{\mathsf{A} : a\#N^{\sharp}\}$, with $N^{\sharp} \in \{\top, \bot\}$, and $\top$ denoting any valid secret value. The secret commitment action is changed accordingly. While being a restrictive assumption, since it limits the expressivity of the BitML language, it heavily simplifies the treatment and the analysis of the correctness. We will try to relax this assumption in a future work. To address *(ii)*, we redefine the transition rules to prevent the progression of time after all the timeouts in C, if any, have expired. Formally, let $ticks(C)$ be the set of all timeouts occurring in expressions of the form $\texttt{after}\, t\colon D$ in contract C, and let $t_{\max}$ the largest of such timeouts. Then, in $\mathcal{G}^{\sharp}_{\{G\}C}$, we impose that if $\mathrm{Time}(s) = t_{\max}$, the action $delay$ from s does not increase the time. Note that we can also simplify the time progression machinery whenever this condition is triggered, but it is inconsequential to the correctness of the abstraction. Intuitively, $\mathcal{G}^{\sharp}_{\{G\}C}$ is the same as $\mathcal{G}_{\{G\}C}$ except that the participants can only decide whether they want to commit to a secret or not, and that time cannot increase indefinitely. It is easy to see that, $\mathcal{G}^{\sharp}_{\{G\}C}$ is a sound and complete abstraction: under the assumption on `put-reveal-if`-clauses, the actual value of a valid secret does not matter for the execution of the contract, and the evaluation of propositions in $\mathrm{AP}_{\{G\}C}$, do not depend on time after $\mathrm{Time}(s) > t_{\max}$; hence, the construction of $\mathcal{G}^{\sharp}_{\{G\}C}$ does not affect the truth of the ATL* specification.

Theorem 3. *Under Assumption 1 and choice of propositions* $\mathrm{AP}_{\{G\}C}$, $\mathcal{G}_{\{G\}C} \models_F \varphi$ *iff* $\mathcal{G}^{\sharp}_{\{G\}C} \models_F \varphi$.

Note that a different choice of propositions for $\mathcal{G}^{\sharp}_{\{G\}C}$ does not guarantee the validity of Theorem 3, e.g., propositions that check arbitrary conditions on secret values.

Despite ATL* model checking under imperfect information and perfect recall is undecidable in general [60], it can be shown that $\mathcal{G}^{\sharp}_{\{G\}C}$ can be cast into a game structure with only *public actions* [61], for which model checking against ATL* specifications is decidable. Such games satisfy the condition that if $J \neq J'$ and $s \sim_a s'$ then $\tau(s, J) \not\sim_a \tau(s', J')$. Although all actions must be public, we can still model private update of an agent's private state [84]. Formally, we define $\mathcal{G}^{\sharp,PA}_{\{G\}C}$ as $\mathcal{G}^{\sharp}_{\{G\}C}$ with the following differences: *(i)* S^{PA}_{ι} contains (at most) $2^{|\mathcal{S}|}$ initial states, one for each possible abstract assignment ($\bot$ or $\top$) of each secret; *(ii)* the commitment action is replaced by $toggle_{\mathsf{A}}$, which must be called exactly once; it can either leave a secret value unchanged or change it, e.g., from $\top$ to $\bot$ or vice versa; *(iii)* the state records the last joint action; *(iv)* $(s, J) \sim^{PA}_{a} (s', J')$, iff $s \sim_a s'$ and $J = J'$, for $(s, J), (s', J') \in \mathrm{St}^{PA}$.

Lemma 4. *Under Assumption 1 and choice of propositions* $\mathrm{AP}_{\{G\}C}$, $\mathcal{G}^{\sharp}_{\{G\}C} \models_F \varphi$ *iff* $\mathcal{G}^{\sharp,PA}_{\{G\}C} \models_F \varphi$.

Proof sketch. Given a strategy for one of the two games, we can compute a strategy for the other game such that, after contract initialization, the secrets of the protagonist coalitions are set to the same values in both games. Moreover, note that in both evaluations, all the possible assignments of the adversaries are considered: in one case because $\mathcal{G} \models \varphi$ iff $\forall s_{\iota} \in S_{\iota}.(\mathcal{G}, s_{\iota}) \models \varphi$, while in the other because we use subjective semantics and mutual knowledge as collective knowledge relation. □

Theorem 5 *Under Assumption 1 and choice of propositions* $\mathrm{AP}_{\{G\}C}$, $\mathcal{G}_{\{G\}C} \models_F \varphi$ *is decidable.*

Proof. First, we determine whether $\mathcal{G}^{\sharp,PA}_{\{G\}C} \models_F \varphi$, which is decidable (see Thm. 1 in [61]); then, the result for $\mathcal{G}_{\{G\}C} \models_F \varphi$ follows by applying Lemma 4 and Theorem 3. □

We conjecture that the problem is decidable also without Assumption 1: intuitively, since the contract is finite, there are a finite number of predicates p in `put-reveal-if`-clauses, meaning that there are only finitely many sets of relevant decisions that the participants can make on secret commitments.

6. Use Cases

Liquidity [53, 54, 85] is an important security property of smart contracts. Generally speaking, a liquidity analysis of a smart contract aims to determine whether its balance can be decreased or funds remain frozen and under which conditions these scenarios happen. The notion of liquidity is based upon liquidability; here, we use the variant formalized in [85]. Informally, a BitML contract x is

liquidable by A if, at any point during its execution, A can perform a sequence of transitions eventually leading to a configuration containing no contract names originating from x. 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. In our framework, it can be shown that a BitML contract C is liquid in Γ_0 iff $\mathcal{G}_{\{G\}C} \models \langle\!\langle\ \rangle\!\rangle\Box((\bigwedge_{a\in\mathcal{S}_\mathsf{A}} \mathsf{committed}_a(\top)) \to \langle\!\langle \mathsf{A}\rangle\!\rangle\Diamond \mathsf{liquidated}_C)$. The constraints of the form $\mathsf{committed}_a(N^\sharp)$, meaning that there exists an abstract commitment $N^\sharp$ for secret a in the current configuration, are required because in the original setup of [85] A must be a "honest" participant who always generates valid commitments to her secrets. There are other variants of liquidity [54] that can be formalized in ATL*:

• *Multiparty liquidity.* Instead of a single agent A, we can consider a set $P \subseteq \mathcal{P}$ of collaborative participants [53]. In the ATL* specification, it is enough to replace A with a coalition of participants P.
• *Liquidity under a strategy.* We can allow A to follow a specific strategy since the start of the game. This means the original ATL* liquidity specification can be simplified in $\langle\!\langle \mathsf{A}\rangle\!\rangle\Diamond \mathsf{liquidated}_C$.
• *Liquidity under assumptions.* We can constrain participants' behavior by working at the specification level. For example, to verify liquidity under the requirement that A always performs the reveal of a secret $a \in \mathcal{S}_\mathsf{A}$, we can write: $\langle\!\langle\ \rangle\!\rangle\Box(\mathsf{revealed}_a \to \langle\!\langle \mathsf{A}\rangle\!\rangle \mathsf{liquidated}_C)$.
• *Quantitative liquidity.* In some cases, A could accept that a portion of the funds remain frozen, e.g. when these funds would be assigned to other participants. We define a contract v-*liquid* for A if a strategy exists for A to withdraw at least v bitcoins: $\langle\!\langle \mathsf{A}\rangle\!\rangle\Box(\mathsf{initialized}_C \to \langle\!\langle \mathsf{A}\rangle\!\rangle\Diamond \text{“}\mathsf{total\text{-}deposits}_\mathsf{A} \geq v\text{”})$.

Our framework can express more general strategic specifications. For example, in a *mutual timed commitment* contract [54], A and B have to exchange their secrets or pay a 1฿ penalty. Consider the following BitML contract that specifies the mutual timed commitment (with $t < t'$):

$$\texttt{reveal}\ a.(\texttt{reveal}\ b.\texttt{split}(1฿ \to \texttt{withdraw}\ \mathsf{A} \mid 1฿ \to \texttt{withdraw}\ \mathsf{B}) + \texttt{after}\ t' : \texttt{withdraw}\ \mathsf{A}) + \texttt{after}\ t : \texttt{withdraw}\ \mathsf{B}$$

We can check some strategic invariants, e.g. if the secret a is not valid, A cannot recover her funds: $\langle\!\langle\ \rangle\!\rangle\Box((\mathsf{committed}_a(\bot) \to \neg\langle\!\langle \mathsf{A}\rangle\!\rangle\Diamond \text{“}\mathsf{total\text{-}deposits}_\mathsf{A} > 0\text{”})$; or that, if timeout t has expired and secret a has not been revealed, then B can recover his funds plus A's penalty without revealing its secret b: $\langle\!\langle\ \rangle\!\rangle\Box((\neg\mathsf{revealed}_b \wedge \mathsf{committed}_a(\bot) \wedge \mathsf{expired}_t) \to \langle\!\langle \mathsf{B}\rangle\!\rangle\Diamond(\neg\mathsf{revealed}_b \wedge \text{“}\mathsf{total\text{-}deposits}_\mathsf{B} = 2\text{”}))$.

Let us consider the following escrow contract *Escrow* between two participants A and B, where the precondition requires A to deposit 1฿ [54]:

$$\mathit{Escrow} = \mathsf{A} : \texttt{withdraw}\ \mathsf{B} + \mathsf{B} : \texttt{withdraw}\ \mathsf{A} + \mathsf{A} : \mathit{Resolve} + \mathsf{B} : \mathit{Resolve}$$
$$\mathit{Resolve} = \texttt{split}(0.1฿ \to \texttt{withdraw}\ \mathsf{M} \mid 0.9฿ \to \mathsf{M} : \texttt{withdraw}\ \mathsf{A} + \mathsf{M} : \texttt{withdraw}\ \mathsf{B})$$

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 1฿ deposit is split in two parts: 0.1฿ go to the mediator, while 0.9฿ are assigned either to A and B, depending on M's choice. We have that $\mathcal{G}_{\{G\}C} \not\models \langle\!\langle A\rangle\!\rangle\Diamond \mathsf{liquidated}_{\mathit{Escrow}}$, for any $A \in \mathcal{P}$, but $\mathcal{G}_{\{G\}C} \models \langle\!\langle P\rangle\!\rangle\Diamond \mathsf{liquidated}_{\mathit{Escrow}}$ for $P \subseteq \mathcal{P}$ with $|P| \geq 2$, i.e. with at least two cooperative participants, the contract is liquid.

7. Conclusion

Our technique makes it possible to reason about several strategic abilities of participants of a Bitcoin smart contract under a unifying framework based on ATL* model checking. Indeed, as we have shown, interesting variants of contract liquidity proposed in the literature can be captured by ATL* specifications. Moreover, as we have seen with the mutual time commitment and the escrow contracts, more general strategic interaction can be captured. We proved that the model checking problem for BitML contracts, under reasonable assumptions, is decidable even in the case of perfect recall and imperfect information semantics. There are several future research directions. First, we would like to extend our approach to the analysis of *rational* players [86] and to quantitative objectives [87, 88, 89]. Then, it would be

interesting to analyze the impact of different collective knowledge relations (mutual, common, and distributed knowledge) by considering some form of communication mechanisms [83] or information sharing (e.g. via *resolution operators* [90]). Moreover, we would like to devise a solution method for general BitML contracts (i.e. without Assumption 1) and provide a tool that solves our problem, relying on state-of-the-art tools such as MCMAS [68], or STV [91] with support for asynchronous models [76, 77, 92].

References

[1] D. Aineto, R. De Benedictis, M. Maratea, M. Mittelmann, G. Monaco, E. Scala, L. Serafini, I. Serina, F. Spegni, E. Tosello, A. Umbrico, M. Vallati (Eds.), Proceedings of the International Workshop on Artificial Intelligence for Climate Change, the Italian workshop on Planning and Scheduling, the RCRA Workshop on Experimental evaluation of algorithms for solving problems with combinatorial explosion, and the Workshop on Strategies, Prediction, Interaction, and Reasoning in Italy (AI4CC-IPS-RCRA-SPIRIT 2024), co-located with 23rd International Conference of the Italian Association for Artificial Intelligence (AIxIA 2024), CEUR Workshop Proceedings, CEUR-WS.org, 2024.

[2] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system (2008). URL: https://bitcoin.org/bitcoin.pdf, https://bitcoin.org/bitcoin.pdf.

[3] V. Buterin, Ethereum: A next-generation smart contract and decentralized application platform (2014). URL: https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf.

[4] N. Szabo, Formalizing and securing relationships on public networks, First Monday 2 (1997). URL: https://firstmonday.org/ojs/index.php/fm/article/view/548/469.

[5] N. Szabo, The Idea of Smart Contracts, 1997. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html.

[6] Script - Bitcoin Wiki, ???? URL: https://en.bitcoin.it/wiki/Script.

[7] Contract - Bitcoin Wiki, 2012. URL: https://en.bitcoin.it/wiki/Contract, https://en.bitcoin.it/wiki/Contract.

[8] N. Atzei, M. Bartoletti, T. Cimoli, S. Lande, R. Zunino, Sok: Unraveling bitcoin smart contracts, in: POST, volume 10804 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 217–242.

[9] M. Andrychowicz, S. Dziembowski, D. Malinowski, L. Mazurek, Fair two-party computations via bitcoin deposits, in: Financial Cryptography Workshops, volume 8438 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 105–121.

[10] M. Bartoletti, R. Zunino, Constant-deposit multiparty lotteries on bitcoin, in: Financial Cryptography Workshops, volume 10323 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 231–247.

[11] S. Uchizono, T. Nakai, Y. Watanabe, M. Iwamoto, Constant-deposit multiparty lotteries on bitcoin for arbitrary number of players and winners, in: ICISC (2), volume 14562 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 133–156.

[12] I. Bentov, R. Kumaresan, How to use bitcoin to design fair protocols, in: CRYPTO (2), volume 8617 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 421–439.

[13] A. Miller, I. Bentov, Zero-collateral lotteries in bitcoin and ethereum, in: EuroS&P Workshops, IEEE, 2017, pp. 4–13.

[14] R. Kumaresan, T. Moran, I. Bentov, How to use bitcoin to play decentralized poker, in: CCS, ACM, 2015, pp. 195–206.

[15] C. Decker, R. Wattenhofer, A fast and scalable payment network with bitcoin duplex micropayment channels, in: SSS, volume 9212 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 3–18.

[16] S. Delgado-Segura, C. Pérez-Solà, G. Navarro-Arribas, J. Herrera-Joancomartí, A fair protocol for data trading based on bitcoin transactions, Future Gener. Comput. Syst. 107 (2020) 832–840.

[17] J. Poon, T. Dryja, The bitcoin lightning network: Scalable off-chain instant payments (2016). https://lightning.network/lightning-network-paper.pdf.
[18] W. Banasik, S. Dziembowski, D. Malinowski, Efficient zero-knowledge contingent payments in cryptocurrencies without scripts, in: ESORICS (2), volume 9879 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 261–280.
[19] M. Andrychowicz, S. Dziembowski, D. Malinowski, L. Mazurek, Secure multiparty computations on bitcoin, Commun. ACM 59 (2016) 76–84.
[20] R. Kumaresan, I. Bentov, How to use bitcoin to incentivize correct computations, in: CCS, ACM, 2014, pp. 30–41.
[21] D. Siegel, Understanding The DAO Attack, 2016. URL: https://www.coindesk.com/learn/understanding-the-dao-attack/.
[22] L. Breidenbach, P. Daian, A. Juels, E. Sirer, An In-Depth Look at the Parity Multisig Bug, ???? URL: https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/.
[23] P. Technologies, A postmortem on the parity multi-sig library self-destruct (2017). URL: https://web.archive.org/web/20231106123811/https://www.parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct/.
[24] K. Nelaturu, A. Mavridou, A. G. Veneris, A. Laszka, Verified development and deployment of multiple interacting smart contracts with verisolid, in: IEEE ICBC, IEEE, 2020, pp. 1–9.
[25] King of ether throne post-mortem investigation, 2016. URL: https://www.kingoftheether.com/postmortem.html, https://www.kingoftheether.com/postmortem.html.
[26] G. Bigi, A. Bracciali, G. Meacci, E. Tuosto, Validation of decentralised smart contracts through game theory and formal methods, in: Programming Languages with Applications to Biology and Security, volume 9465 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 142–161.
[27] K. Chatterjee, A. K. Goharshady, Y. Velner, Quantitative analysis of smart contracts, in: ESOP, volume 10801 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 739–767.
[28] C. Laneve, C. S. Coen, A. Veschetti, On the prediction of smart contracts' behaviours, in: From Software Engineering to Formal Methods and Tools, and Back, volume 11865 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 397–415.
[29] R. van der Meyden, On the specification and verification of atomic swap smart contracts (extended abstract), in: IEEE ICBC, IEEE, 2019, pp. 176–179.
[30] K. Delmolino, M. Arnett, A. E. Kosba, A. Miller, E. Shi, Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab, in: Financial Cryptography Workshops, volume 9604 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 79–94.
[31] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on ethereum smart contracts (sok), in: POST, volume 10204 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 164–186.
[32] A. Miller, Z. Cai, S. Jha, Smart contracts and opportunities for formal methods, in: ISoLA (4), volume 11247 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 280–299.
[33] S. Badruddoja, R. Dantu, Y. He, K. Upadhyay, M. A. Thompson, Making smart contracts smarter, in: IEEE ICBC, IEEE, 2021, pp. 1–3.
[34] R. Feichtinger, R. Fritsch, L. Heimbach, Y. Vonlanthen, R. Wattenhofer, Sok: Attacks on daos, in: The 4th Workshop on Decentralized Finance (DeFi), 2024.
[35] A. Groce, J. Feist, G. Grieco, M. Colburn, What are the actual flaws in important smart contracts (and how can we find them)?, in: Financial Cryptography, volume 12059 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 634–653.
[36] A. Mense, M. Flatscher, Security vulnerabilities in ethereum smart contracts, in: iiWAS, ACM, 2018, pp. 375–380.
[37] S. Sayeed, H. Marco-Gisbert, T. Caira, Smart contract: Attacks and protections, IEEE Access 8 (2020) 24416–24427.
[38] D. Harz, W. J. Knottenbelt, Towards safer smart contracts: A survey of languages and verification methods, CoRR abs/1809.09805 (2018).
[39] A. Singh, R. M. Parizi, Q. Zhang, K. R. Choo, A. Dehghantanha, Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities, Comput. Secur. 88 (2020).

[40] P. Tolmach, Y. Li, S. Lin, Y. Liu, Z. Li, A survey of smart contract formal specification and verification, ACM Comput. Surv. 54 (2022) 148:1–148:38.
[41] R. B. Fekih, M. Lahami, M. Jmaiel, S. Bradai, Formal verification of smart contracts based on model checking: An overview, in: WETICE, IEEE, 2023, pp. 1–6.
[42] E. M. Clarke, E. A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, in: Logic of Programs, volume 131 of *Lecture Notes in Computer Science*, Springer, 1981, pp. 52–71.
[43] J. P. Queille, J. Sifakis, Specification and verification of concurrent systems in cesar, in: M. Dezani-Ciancaglini, U. Montanari (Eds.), International Symposium on Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 1982, pp. 337–351.
[44] E. M. Clarke, O. Grumberg, D. A. Peled, Model checking, 1st Edition, MIT Press, 2001.
[45] C. Baier, J. Katoen, Principles of model checking, MIT Press, 2008.
[46] S. Kripke, Semantical considerations on modal logic, Acta Philosophica Fennica 16 (1963) 83–94.
[47] R. M. Keller, Formal verification of parallel programs, Commun. ACM 19 (1976) 371–384.
[48] A. Pnueli, The temporal logic of programs, in: FOCS, IEEE Computer Society, 1977, pp. 46–57.
[49] E. A. Emerson, J. Y. Halpern, "sometimes" and "not never" revisited: on branching versus linear time temporal logic, J. ACM 33 (1986) 151–178.
[50] D. Harel, A. Pnueli, On the development of reactive systems, in: Logics and Models of Concurrent Systems, volume 13 of *NATO ASI Series*, Springer, 1984, pp. 477–498.
[51] O. Kupferman, M. Y. Vardi, P. Wolper, Module checking, Inf. Comput. 164 (2001) 322–344.
[52] M. Bartoletti, Smart contracts contracts, Frontiers Blockchain 3 (2020) 27.
[53] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, M. T. Vechev, Securify: Practical security analysis of smart contracts, in: CCS, ACM, 2018, pp. 67–82.
[54] M. Bartoletti, R. Zunino, Verifying liquidity of bitcoin contracts, in: POST, volume 11426 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 222–247.
[55] W. Jamroga, A. Murano, Module checking of strategic ability, in: AAMAS, ACM, 2015, pp. 227–235.
[56] Y. Shoham, K. Leyton-Brown, Multiagent Systems - Algorithmic, Game-Theoretic, and Logical Foundations, Cambridge University Press, 2009.
[57] M. J. Wooldridge, An Introduction to MultiAgent Systems, Second Edition, Wiley, 2009.
[58] M. Bartoletti, R. Zunino, Bitml: A calculus for bitcoin smart contracts, in: CCS, ACM, 2018, pp. 83–100.
[59] R. Alur, T. A. Henzinger, O. Kupferman, Alternating-time temporal logic, J. ACM 49 (2002) 672–713.
[60] C. Dima, F. L. Tiplea, Model-checking ATL under imperfect information and perfect recall semantics is undecidable, CoRR abs/1102.4225 (2011).
[61] F. Belardinelli, A. Lomuscio, A. Murano, S. Rubin, Verification of multi-agent systems with imperfect information and public actions, in: AAMAS, ACM, 2017, pp. 1268–1276.
[62] K. Crary, M. J. Sullivan, Peer-to-peer affine commitment using bitcoin, in: PLDI, ACM, 2015, pp. 479–488.
[63] N. Atzei, M. Bartoletti, S. Lande, N. Yoshida, R. Zunino, Developing secure bitcoin contracts with bitml, in: ESEC/SIGSOFT FSE, ACM, 2019, pp. 1124–1128.
[64] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, J. F. Quesada, Maude: specification and programming in rewriting logic, Theor. Comput. Sci. 285 (2002) 187–243.
[65] S. Eker, J. Meseguer, A. Sridharanarayanan, The maude LTL model checker, in: WRLA, volume 71 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2002, pp. 162–187.
[66] G. Madl, L. A. D. Bathen, G. H. Flores, D. Jadav, Formal verification of smart contracts using interface automata, in: Blockchain, IEEE, 2019, pp. 556–563.
[67] W. Nam, H. Kil, Formal verification of blockchain smart contracts via ATL model checking, IEEE Access 10 (2022) 8151–8162.
[68] A. Lomuscio, H. Qu, F. Raimondi, MCMAS: an open-source model checker for the verification of multi-agent systems, Int. J. Softw. Tools Technol. Transf. 19 (2017) 9–30.
[69] M. Bartoletti, T. Cimoli, R. Zunino, Fun with bitcoin smart contracts, in: ISoLA (4), volume 11247 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 432–449.

[70] S. Busard, C. Pecheur, H. Qu, F. Raimondi, Reasoning about memoryless strategies under partial observability and unconditional fairness constraints, Inf. Comput. 242 (2015) 128–156.
[71] N. Bulling, W. Jamroga, Comparing variants of strategic ability: how uncertainty and memory influence general properties of games, Auton. Agents Multi Agent Syst. 28 (2014) 474–518.
[72] R. Fagin, J. Y. Halpern, Y. Moses, M. Y. Vardi, Reasoning About Knowledge, MIT Press, 1995.
[73] X. Nicollin, J. Sifakis, An overview and synthesis on timed process algebras, in: CAV, volume 575 of *Lecture Notes in Computer Science*, Springer, 1991, pp. 376–398.
[74] A. Meski, W. Penczek, M. Szreter, B. Wozna-Szczesniak, A. Zbrzezny, Bdd-versus sat-based bounded model checking for the existential fragment of linear temporal logic with knowledge: algorithms and their performance, Auton. Agents Multi Agent Syst. 28 (2014) 558–604.
[75] A. Lomuscio, W. Penczek, H. Qu, Partial order reductions for model checking temporal-epistemic logics over interleaved multi-agent systems, Fundam. Informaticae 101 (2010) 71–90.
[76] W. Jamroga, W. Penczek, T. Sidoruk, P. Dembinski, A. W. Mazurkiewicz, Towards partial order reductions for strategic ability, J. Artif. Intell. Res. 68 (2020) 817–850.
[77] W. Jamroga, W. Penczek, T. Sidoruk, Strategic abilities of asynchronous agents: Semantic side effects and how to tame them, in: KR, 2021, pp. 368–378.
[78] F. Laroussinie, N. Markey, G. Oreiby, On the expressiveness and complexity of ATL, Log. Methods Comput. Sci. 4 (2008).
[79] P. Schobbens, Alternating-time logic with imperfect recall, in: LCMAS, volume 85 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2003, pp. 82–93.
[80] W. Jamroga, W. van der Hoek, Agents that know how to play, Fundam. Informaticae 63 (2004) 185–219.
[81] N. Bulling, W. Jamroga, M. Popovici, Reasoning about strategic abilities: Agents with truly perfect recall, ACM Trans. Comput. Log. 20 (2019) 10:1–10:46.
[82] D. P. Guelev, C. Dima, Model-checking strategic ability and knowledge of the past of communicating coalitions, in: DALT, volume 5397 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 75–90.
[83] C. Dima, C. Enea, D. P. Guelev, Model-checking an alternating-time temporal logic with knowledge, imperfect information, perfect recall and communicating coalitions, in: GANDALF, volume 25 of *EPTCS*, 2010, pp. 103–117.
[84] F. Belardinelli, A. Lomuscio, A. Murano, S. Rubin, Verification of multi-agent systems with public actions against strategy logic, Artif. Intell. 285 (2020) 103302.
[85] M. Bartoletti, S. Lande, M. Murgia, R. Zunino, Verifying liquidity of recursive bitcoin contracts, Log. Methods Comput. Sci. 18 (2022).
[86] A. Abate, J. Gutierrez, L. Hammond, P. Harrenstein, M. Kwiatkowska, M. Najib, G. Perelli, T. Steeples, M. J. Wooldridge, Rational verification: game-theoretic verification of multi-agent systems, Appl. Intell. 51 (2021) 6569–6584.
[87] N. Bulling, V. Goranko, Combining quantitative and qualitative reasoning in concurrent multi-player games, Auton. Agents Multi Agent Syst. 36 (2022) 2.
[88] P. Bouyer, O. Kupferman, N. Markey, B. Maubert, A. Murano, G. Perelli, Reasoning about quality and fuzziness of strategic behaviors, ACM Trans. Comput. Log. 24 (2023) 21:1–21:38.
[89] W. Jamroga, M. Mittelmann, A. Murano, G. Perelli, Playing quantitative games against an authority: On the module checking problem, in: AAMAS, International Foundation for Autonomous Agents and Multiagent Systems / ACM, 2024, pp. 926–934.
[90] T. Ågotnes, Y. N. Wáng, Resolving distributed knowledge, Artif. Intell. 252 (2017) 1–21.
[91] D. Kurpiewski, W. Jamroga, M. Knapik, STV: model checking for strategies under imperfect information, in: AAMAS, International Foundation for Autonomous Agents and Multiagent Systems, 2019, pp. 2372–2374.
[92] D. Kurpiewski, W. Pazderski, W. Jamroga, Y. Kim, Stv+reductions: Towards practical verification of strategic ability using model reductions, in: AAMAS, ACM, 2021, pp. 1770–1772.