A True Concurrent Model of Smart Contracts Executions

Massimo Bartoletti, Letterio Galletta, Maurizio Murgia

▶ **To cite this version:**

Massimo Bartoletti, Letterio Galletta, Maurizio Murgia. A True Concurrent Model of Smart Contracts Executions. 22th International Conference on Coordination Languages and Models (COORDINATION), Jun 2020, Valletta, Malta. pp.243-260, 10.1007/978-3-030-50029-0_16 . hal-03273986

HAL Id: hal-03273986
https://inria.hal.science/hal-03273986

Submitted on 29 Jun 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

A true concurrent model of smart contracts executions

Massimo Bartoletti[1][0000−0003−3796−9774], Letterio Galletta[2][0000−0003−0351−9169], and Maurizio Murgia[3][0000−0001−7613−621X]

[1] University of Cagliari, Italy bart@unica.it
[2] IMT School for Advanced Studies, Lucca, Italy letterio.galletta@imtlucca.it
[3] University of Trento, Italy maurizio.murgia@unitn.it

Abstract. The development of blockchain technologies has enabled the trustless execution of so-called *smart contracts*, i.e. programs that regulate the exchange of assets (e.g., cryptocurrency) between users. In a decentralized blockchain, the state of smart contracts is collaboratively maintained by a peer-to-peer network of mutually untrusted nodes, which collect from users a set of *transactions* (representing the required actions on contracts), and execute them in some order. Once this sequence of transactions is appended to the blockchain, the other nodes validate it, re-executing the transactions in the same order. The serial execution of transactions does not take advantage of the multi-core architecture of modern processors, so contributing to limit the throughput. In this paper we propose a true concurrent model of smart contracts execution. Based on this, we show how static analysis of smart contracts can be exploited to parallelize the execution of transactions.

1 Introduction

Smart contracts [21] are computer programs that transfer digital assets between users without a trusted authority. Currently, smart contracts are supported by several blockchains, the first and most widespread one being Ethereum [9]. Users interact with a smart contract by sending *transactions*, which trigger state updates, and may possibly involve transfers of crypto-assets between the called contract and the users. The sequence of transactions on the blockchain determines the state of each contract, and the balance of each user.

The blockchain is maintained by a peer-to-peer network of nodes, which follow a consensus protocol to determine, at each turn, a new block of transactions to be added to the blockchain. This protocol guarantees the correct execution of contracts also in the presence of (a minority of) adversaries in the network, and ensures that all the nodes have the same view of their state. Nodes play the role of *miner* or that of *validator*. Miners gather from the network sets of transactions sent by users, and execute them *serially* to determine the new state. Once a block is appended to the blockchain, validators re-execute all its transactions, to update their local view of the contracts state and of the users' balance. To do this, validators process the transactions exactly in the same order in which they

occur in the block, since choosing a different order could potentially result in inconsistencies between the nodes (note that miners also act as validators, since they validate all the blocks received from the network).

Although executing transactions in a purely sequential fashion is quite effective to ensure the consistency of the blockchain state, in the age of multi-core processors it fails to properly exploit the computational capabilities of nodes. By enabling miners and validators to concurrently execute transactions, it would be possible to improve the efficiency and the throughput of the blockchain.

This paper exploits techniques from concurrency theory to provide a formal backbone for parallel executions of transactions. More specifically, our main contributions can be summarised as follows:

- As a first step, we formalise blockchains, giving their semantics as a function which maps each contract to its state, and each user to her balance. This semantics reflects the standard implementation of nodes, where transactions are evaluated in sequence, without any concurrency.
- We introduce two notions of *swappability* of transactions. The first is purely semantic: two adjacent transactions can be swapped if doing so preserves the semantics of the blockchain. The second notion, called *strong* swappability, is more syntactical: it checks a simple condition (inspired by Bernstein's conditions [7]) on static approximations of the variables read/written by the transactions. Theorem 2 shows that strong swappability is strictly included in the semantic relation. Further, if we transform a blockchain by repeatedly exchanging adjacent strongly swappable transactions, the resulting blockchain is observationally equivalent to the original one (Theorem 4).
- Building upon strong swappability, we devise a true concurrent model of transactions execution. To this purpose, we transform a block of transactions into an *occurrence net*, describing exactly the partial order induced by the swappability relation. We model the concurrent executions of a blockchain in terms of the *step firing sequences* (i.e. finite sequences of *sets* of transitions) of the associated occurrence net. Theorem 5 establishes that the concurrent executions and the serial one are semantically equivalent.
- We describe how miners and validators can use our results to concurrently execute transactions, exploiting the multi-core architecture available on their nodes. Remarkably, our technique is compatible with the current implementation of the Ethereum blockchain, while the other existing approaches to parallelize transactions execution would require a soft-fork.
- We apply our technique to ERC-721 tokens, one of the most common kinds of contracts in Ethereum, showing them to be suitable for parallelization.

Because of space constraints, all the proofs of our results are in [6].

2 Transactions and blockchains

In this section we introduce a general model of transactions and blockchains, abstracting from the actual smart contracts language.

A smart contract is a finite set of functions, i.e. terms of the form $\mathtt{f}(\boldsymbol{x})\{S\}$, where $\mathtt{f}$ is a function name, $\boldsymbol{x}$ is the sequence of formal parameters (omitted when empty), and S is the function body. We postulate that the functions in a contract have distinct names. We abstract from the actual syntax of S, and we just assume that the semantics of function bodies is defined (see e.g. [5] for a concrete instance of syntax and semantics of function bodies).

Let **Val** be a set of *values*, ranged over by $v, v', \ldots$, let **Const** be a set of *constant names* $x, y, \ldots$, and let **Addr** be a set of *addresses* $\mathcal{X}, \mathcal{Y}, \ldots$, partitioned into *account addresses* $\mathcal{A}, \mathcal{B}, \ldots$ and *contract addresses* $\mathcal{C}, \mathcal{D}, \ldots$. We assume a mapping Γ from addresses to contracts.

We assume that each contract has a key-value store, which we render as a partial function $\mathbf{Val} \rightharpoonup \mathbf{Val}$ from keys $k \in \mathbf{Val}$ to values. The state of the blockchain is a function $\sigma : \mathbf{Addr} \to (\mathbf{Val} \rightharpoonup \mathbf{Val})$ from addresses to key-value stores. We postulate that $\mathtt{balance} \in \mathrm{dom}\,\sigma\mathcal{X}$ for all $\mathcal{X}$. A *qualified key* is a term of the form $\mathcal{X}.k$. We write $\sigma(\mathcal{X}.k)$ for $\sigma\mathcal{X}k$; when $k \notin \mathrm{dom}\,\sigma\,\mathcal{X}$, we write $\sigma\,(\mathcal{X}.k) = \bot$. We use $p, q, \ldots$ to range over qualified keys, $P, Q, \ldots$ to range over sets of them, and $\mathbb{P}$ to denote the set of all qualified keys.

To have a uniform treatment of accounts and contracts, we assume that for all account addresses $\mathcal{A}$, $\mathrm{dom}\,\sigma\mathcal{A} = \{\mathtt{balance}\}$, and that the contract $\Gamma(\mathcal{A})$ has exactly one function, which just skips. In this way, the statement $\mathcal{A}.\mathtt{transfer}(n)$, which transfers n currency units to $\mathcal{A}$, can be rendered as a call to this function.

State updates define how values associated with qualified keys are modified.

Definition 1 (State update). *A* state update $\pi : \mathbf{Addr} \rightharpoonup (\mathbf{Val} \rightharpoonup \mathbf{Val})$ *is a function from qualified keys to values; we denote with* $\{v/\mathcal{X}.k\}$ *the state update which maps* $\mathcal{X}.k$ *to* v*. We define* $\mathrm{keys}(\pi)$ *as the set of qualified keys* $\mathcal{X}.k$ *such that* $\mathcal{X} \in \mathrm{dom}\,\pi$ *and* $k \in \mathrm{dom}\,\pi\mathcal{X}$*. We apply updates to states as follows:*

$$(\sigma\pi)\mathcal{X} = \delta_{\mathcal{X}} \quad \textit{where} \quad \delta_{\mathcal{X}}\,k = \begin{cases} \pi\mathcal{X}k & \textit{if } \mathcal{X}.k \in \mathrm{keys}(\pi) \\ \sigma\mathcal{X}k & \textit{otherwise} \end{cases} \qquad \diamond$$

We denote with $[\![S]\!]^{\mathcal{X}}_{\sigma,\rho}$ the semantics of the statement S. This semantics is either a blockchain state σ', or it is undefined (denoted by $\bot$). The semantics is parameterised over a state σ, an address $\mathcal{X}$ (the contract wherein S is evaluated), and an *environment* $\rho : \mathbf{Const} \rightharpoonup \mathbf{Val}$, used to evaluate the formal parameters and the special names $\mathtt{sender}$ and $\mathtt{value}$. These names represent, respectively, the caller of the function, and the amount of currency transferred along with the call. We postulate that $\mathtt{sender}$ and $\mathtt{value}$ are not used as formal parameters.

We define the auxiliary operators $+$ and $-$ on states as follows:

$$\sigma \circ (\mathcal{X} : n) \;=\; \sigma\{{}^{(\sigma\mathcal{X}\mathtt{balance}) \circ n}/_{\mathcal{X}.\mathtt{balance}}\} \qquad (\circ \in \{+, -\})$$

i.e., $\sigma + \mathcal{X} : n$ updates σ by increasing the $\mathtt{balance}$ of $\mathcal{X}$ of n currency units.

A *transaction* T is a term of the form:

$$\mathcal{A} \xrightarrow{n} \mathcal{C} : \mathtt{f}(\boldsymbol{v})$$

$$\frac{\mathtt{f}(\boldsymbol{x})\{S\} \in \Gamma(\mathcal{C}) \quad \sigma\,\mathcal{A}\,\mathtt{balance} \geq n \quad [\![S]\!]^{\mathcal{C}}_{\sigma-\mathcal{A}:n+\mathcal{C}:n,\,\{\mathcal{A}/\mathtt{sender},n/\mathtt{value},\boldsymbol{v}/\boldsymbol{x}\}} = \sigma'}{[\![\mathcal{A} \xrightarrow{n} \mathcal{C} : \mathtt{f}(\boldsymbol{v})]\!]_\sigma = \sigma'}\ [\text{Tx1}] \qquad \frac{\mathtt{f}(\boldsymbol{x})\{S\} \in \Gamma(\mathcal{C}) \quad \left(\sigma\,\mathcal{A}\,\mathtt{balance} < n \quad \text{or} \quad [\![S]\!]^{\mathcal{C}}_{\sigma-\mathcal{A}:n+\mathcal{C}:n,\,\{\mathcal{A}/\mathtt{sender},n/\mathtt{value},\boldsymbol{v}/\boldsymbol{x}\}} = \bot\right)}{[\![\mathcal{A} \xrightarrow{n} \mathcal{C} : \mathtt{f}(\boldsymbol{v})]\!]_\sigma = \sigma}\ [\text{Tx2}]$$

Fig. 1: Semantics of transactions.

Intuitively, $\mathcal{A}$ is the address of the caller, $\mathcal{C}$ is the address of the called contract, $\mathtt{f}$ is the called function, n is the value transferred from $\mathcal{A}$ to $\mathcal{C}$, and $\boldsymbol{v}$ is the sequence of actual parameters. We denote the semantics of T in σ as $[\![\mathsf{T}]\!]_\sigma$, where the function $[\![\cdot]\!]_\sigma$ is defined in Figure 1, which we briefly comment.

The semantics of a transaction $\mathsf{T} = \mathcal{A} \xrightarrow{n} \mathcal{C} : \mathtt{f}(\boldsymbol{v})$, in a given blockchain state σ, is a new state σ'. Rule [Tx1] handles the case where the transaction is successful: this happens when $\mathcal{A}$'s balance is at least n, and the function call terminates in a non-error state. Note that n units of currency are transferred to $\mathcal{C}$ *before* starting to execute $\mathtt{f}$, and that the names `sender` and `value` are bound, respectively, to $\mathcal{A}$ and n. Rule [Tx2] applies either when $\mathcal{A}$'s balance is not enough, or the execution of $\mathtt{f}$ fails. In these cases, T does not alter the state.

A *blockchain* $\mathbf{B}$ is a finite sequence of transactions; we denote with ϵ the empty blockchain. The semantics of a blockchain is obtained by folding the semantics of its transactions, starting from a given state σ:

$$[\![\epsilon]\!]_\sigma = \sigma \qquad [\![\mathsf{T}\mathbf{B}]\!]_\sigma = [\![\mathbf{B}]\!]_{[\![\mathsf{T}]\!]_\sigma}$$

Note that erroneous transactions can occur within a blockchain, but they have no effect on its semantics (as rule [Tx2] makes them identities w.r.t. the append operation). We assume that in the initial state of the blockchain, denoted by $\sigma^\star$, each address $\mathcal{X}$ has a balance $n^\star_{\mathcal{X}} \geq 0$, while all the other keys are unbound.

We write $[\![\mathbf{B}]\!]$ for $[\![\mathbf{B}]\!]_{\sigma^\star}$, where $\sigma^\star\mathcal{X} = \{n^\star_{\mathcal{X}}/\mathtt{balance}\}$. We say that a state σ is *reachable* if $\sigma = [\![\mathbf{B}]\!]$ for some $\mathbf{B}$.

Example 1. Consider the following functions of a contract at address $\mathcal{C}$:

$$\mathtt{f}_0()\{x{:=}1\} \qquad \mathtt{f}_1()\{\mathtt{if}\ x = 0\ \mathtt{then}\ \mathcal{B}.\mathtt{transfer}(1)\} \qquad \mathtt{f}_2()\{\mathcal{B}.\mathtt{transfer}(1)\}$$

Let σ be a state such that $\sigma\mathcal{A}\mathtt{balance} \geq 2$, and let $\mathbf{B} = \mathsf{T}_0\mathsf{T}_1\mathsf{T}_2$, where:

$$\mathsf{T}_0 = \mathcal{A} \xrightarrow{0} \mathcal{C} : \mathtt{f}_0() \qquad \mathsf{T}_1 = \mathcal{A} \xrightarrow{1} \mathcal{C} : \mathtt{f}_1() \qquad \mathsf{T}_2 = \mathcal{A} \xrightarrow{1} \mathcal{C} : \mathtt{f}_2()$$

By applying rule [Tx1] three times, we have that:

$$\begin{aligned}
[\![\mathsf{T}_0]\!]_\sigma &= [\![x{:=}1]\!]^{\mathcal{C}}_{\sigma,\,\{\mathcal{A}/\mathtt{sender},0/\mathtt{value}\}} = \sigma\{1/\mathcal{C}.x\} = \sigma' \\
[\![\mathsf{T}_1]\!]_{\sigma'} &= [\![\mathtt{if}\ x = 0\ \mathtt{then}\ \mathcal{B}.\mathtt{transfer}(1)]\!]^{\mathcal{C}}_{\sigma'-\mathcal{A}:1+\mathcal{C}:1,\,\{\mathcal{A}/\mathtt{sender},1/\mathtt{value}\}} \\
&= \sigma' - \mathcal{A} : 1 + \mathcal{C} : 1 = \sigma'' \\
[\![\mathsf{T}_2]\!]_{\sigma''} &= [\![\mathcal{B}.\mathtt{transfer}(1)]\!]^{\mathcal{C}}_{\sigma''-\mathcal{A}:1+\mathcal{C}:1,\,\{\mathcal{A}/\mathtt{sender},1/\mathtt{value}\}} = \sigma'' - \mathcal{A} : 1 + \mathcal{B} : 1
\end{aligned}$$

Summing up, $[\![\mathbf{B}]\!]_\sigma = \sigma\{1/\mathcal{C}.x\} - \mathcal{A} : 2 + \mathcal{B} : 1 + \mathcal{C} : 1$. ◇

3 Swapping transactions

We define two blockchain states to be *observationally equivalent* when they agree on the values associated to all the qualified keys. Our formalisation is parameterised on a set of qualified keys P over which we require the agreement.

Definition 2 (Observational equivalence). *For all $P \subseteq \mathbb{P}$, we define $\sigma \sim_P \sigma'$ iff $\forall p \in P : \sigma p = \sigma' p$. We say that σ and σ' are* observationally equivalent*, in symbols $\sigma \sim \sigma'$, when $\sigma \sim_P \sigma'$ holds for all P.* ◇

Lemma 1. *For all $P, Q \subseteq \mathbb{P}$: (i) $\sim_P$ is an equivalence relation; (ii) if $\sigma \sim_P \sigma'$ and $Q \subseteq P$, then $\sigma \sim_Q \sigma'$; (iii) $\sim = \sim_{\mathbb{P}}$.* ◇

We extend the equivalence relations above to blockchains, by passing through their semantics. For all P, we define $\mathbf{B} \sim_P \mathbf{B'}$ iff $[\![\mathbf{B}]\!]_\sigma \sim_P [\![\mathbf{B'}]\!]_\sigma$ holds for all reachable σ (note that all the definitions and results in this paper apply to reachable states, since the unreachable ones do not represent actual contract executions). We write $\mathbf{B} \sim \mathbf{B'}$ when $\mathbf{B} \sim_P \mathbf{B'}$ holds for all P. The relation $\sim$ is a *congruence* with respect to the append operation, i.e. if $\mathbf{B} \sim \mathbf{B'}$ then we can replace $\mathbf{B}$ with $\mathbf{B'}$ in a larger blockchain, preserving its semantics.

Lemma 2. $\mathbf{B} \sim \mathbf{B'} \implies \forall \mathbf{B}_0, \mathbf{B}_1 : \mathbf{B}_0\mathbf{B}\mathbf{B}_1 \sim \mathbf{B}_0\mathbf{B'}\mathbf{B}_1$. ◇

Two transactions are *swappable* when exchanging their order preserves observational equivalence.

Definition 3 (Swappability). *Two transactions $\mathsf{T} \neq \mathsf{T}'$ are* swappable*, in symbols $\mathsf{T} \rightleftarrows \mathsf{T}'$, when $\mathsf{T}\mathsf{T}' \sim \mathsf{T}'\mathsf{T}$.* ◇

Example 2. Recall the transactions in Example 1. We have that $\mathsf{T}_0 \rightleftarrows \mathsf{T}_2$ and $\mathsf{T}_1 \rightleftarrows \mathsf{T}_2$, but $\mathsf{T}_0 \not\rightleftarrows \mathsf{T}_1$ (see Figure 5 in Appendix A of [6]). ◇

We shall use the theory of trace languages originated from Mazurkiewicz's works [17] to study observational equivalence under various swapping relations. Below, we fix the alphabet of trace languages as the set $\mathbf{Tx}$ of all transactions.

Definition 4 (Mazurkiewicz equivalence). *Let I be a symmetric and irreflexive relation on $\mathbf{Tx}$. The* Mazurkiewicz equivalence $\simeq_I$ *is the least congruence in the free monoid $\mathbf{Tx}^*$ such that: $\forall \mathsf{T}, \mathsf{T}' \in \mathbf{Tx}$: $\mathsf{T}\ I\ \mathsf{T}' \implies \mathsf{T}\mathsf{T}' \simeq_I \mathsf{T}'\mathsf{T}$.*

Theorem 1 below states that the Mazurkiewicz equivalence constructed on the swappability relation $\rightleftarrows$ is an observational equivalence. Therefore, we can transform a blockchain into an observationally equivalent one by a finite number of exchanges of adjacent swappable transactions.

Theorem 1. $\simeq_{\rightleftarrows} \subseteq \sim$. ◇

Example 3. We can rearrange the transactions in Example 1 as $\mathsf{T}_0\mathsf{T}_1\mathsf{T}_2 \sim \mathsf{T}_0\mathsf{T}_2\mathsf{T}_1 \sim \mathsf{T}_2\mathsf{T}_0\mathsf{T}_1$. Instead, $\mathsf{T}_1\mathsf{T}_0\mathsf{T}_2 \not\sim \mathsf{T}_2\mathsf{T}_0\mathsf{T}_1$ (e.g., starting from a state σ such that $\sigma\mathcal{A}\texttt{balance} = 2$ and $\sigma\mathcal{C}x = 0$, see Figure 6 in Appendix A of [6]). ◇

Note that the converse of Theorem 1 does not hold: indeed, $\mathbf{B} \simeq_{\rightleftarrows} \mathbf{B'}$ requires that $\mathbf{B}$ and $\mathbf{B'}$ have the same length, while $\mathbf{B} \sim \mathbf{B'}$ may also hold for blockchains of different length (e.g., $\mathbf{B'} = \mathbf{B}\mathsf{T}$, where T does not alter the state).

Safe approximations of read/written keys Note that the relation $\rightleftarrows$ is undecidable whenever the contract language is Turing-equivalent. So, to detect swappable transactions we follow a static approach, consisting of two steps. First, we over-approximate the set of keys read and written by transactions, by statically analysing the code of the called functions. We then check a simple condition on these approximations (Definition 7), to detect if two transactions can be swapped. Since static analyses to over-approximate read and written variables are quite standard [18], here we just rely on such approximations, by only assuming their safety. In Definition 5 we state that a set P safely approximates the keys *written* by T, when T does not alter the state of the keys not in P. Defining set of *read* keys is a bit trickier: intuitively, we require that if we execute the transaction starting from two states that agree on the values of the keys in the read set, then these executions should be equivalent, in the sense that they do not introduce new differences between the resulting states (with respect to the difference already existing before).

Definition 5 (Safe approximation of read/written keys). *Given a set of qualified keys P and a transaction T, we define:*

$$P \models^w \mathsf{T} \quad \textit{iff} \quad \forall Q : Q \cap P = \emptyset \implies \mathsf{T} \sim_Q \epsilon$$
$$P \models^r \mathsf{T} \quad \textit{iff} \quad \forall \mathbf{B}, \mathbf{B'}, Q : \mathbf{B} \sim_P \mathbf{B'} \wedge \mathbf{B} \sim_Q \mathbf{B'} \implies \mathbf{B}\mathsf{T} \sim_Q \mathbf{B'}\mathsf{T} \quad \diamond$$

Example 4. Let $\mathsf{T} = \mathcal{A} \xrightarrow{1} \mathcal{C} : \mathtt{f}()$, where $\mathtt{f}()\{\mathcal{B}.\mathtt{transfer}(1)\}$ is a function of $\mathcal{C}$. The execution of T affects the `balance` of $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$; however, $\mathcal{C}.\mathtt{balance}$ is first incremented and then decremented, and so its value remains unchanged. Then, $\{\mathcal{A}.\mathtt{balance}, \mathcal{B}.\mathtt{balance}\} \models^w \mathsf{T}$, and it is the smallest safe approximation of the keys written by T. To prove that $P = \{\mathcal{A}.\mathtt{balance}\} \models^r \mathsf{T}$, assume two blockchains $\mathbf{B}$ and $\mathbf{B'}$ and a set of keys Q such that $\mathbf{B} \sim_P \mathbf{B'}$ and $\mathbf{B} \sim_Q \mathbf{B'}$. If $[\![\mathbf{B}]\!]\mathcal{A}\mathtt{balance} < 1$, then by [Tx2] we have $[\![\mathbf{B}\mathsf{T}]\!] = [\![\mathbf{B}]\!]$. Since $\mathbf{B} \sim_P \mathbf{B'}$, then also $[\![\mathbf{B'}]\!]\mathcal{A}\mathtt{balance} < 1$, and so by [Tx2] we have $[\![\mathbf{B'}\mathsf{T}]\!] = [\![\mathbf{B'}]\!]$. Then, $\mathbf{B}\mathsf{T} \sim_Q \mathbf{B'}\mathsf{T}$. Otherwise, if $[\![\mathbf{B}]\!]\mathcal{A}\mathtt{balance} = n \geq 1$, then by [Tx1] the execution of T transfers one unit of currency from $\mathcal{A}$ to $\mathcal{B}$, so the execution of T affects exactly $\mathcal{A}.\mathtt{balance}$ and $\mathcal{B}.\mathtt{balance}$. So, it is enough to show that $\mathbf{B} \sim_{\{q\}} \mathbf{B'}$ implies $\mathbf{B}\mathsf{T} \sim_{\{q\}} \mathbf{B'}\mathsf{T}$ for $q \in \{\mathcal{A}.\mathtt{balance}, \mathcal{B}.\mathtt{balance}\}$. For $q = \mathcal{A}.\mathtt{balance}$, we have that $[\![\mathbf{B'}\mathsf{T}]\!]\mathcal{A}\mathtt{balance} = n - 1 = [\![\mathbf{B}\mathsf{T}]\!]\mathcal{A}\mathtt{balance}$. For $q = \mathcal{B}.\mathtt{balance}$, we have that $[\![\mathbf{B'}\mathsf{T}]\!]\mathcal{B}\mathtt{balance} = [\![\mathbf{B'}]\!]\mathcal{B}\mathtt{balance} + 1 = [\![\mathbf{B}]\!]\mathcal{B}\mathtt{balance} + 1 = [\![\mathbf{B}\mathsf{T}]\!]\mathcal{B}\mathtt{balance}$. Therefore, we conclude that $P \models^r \mathsf{T}$. $\diamond$

Widening a safe approximation (either of read or written keys) preserves its safety; further, the intersection of two safe write approximations is still safe (see Lemma 6 in Appendix A of [6]). From this, it follows that there exists a *least* safe approximation of the keys written by a transaction.

Strong swappability We use safe approximations of the read/written keys to detect when two transactions are swappable. To achieve that, we check whether two transactions T and T' operate on disjoint portions of the blockchain state.

More specifically, we recast in our setting Bernstein's conditions [7] for the parallel execution of processes: it suffices to check that the set of keys written by T is disjoint from those written or read by T', and vice versa. When this happens we say that the two transactions are *strongly swappable*.

Definition 6 (Strong swappability). *We say that two transactions* $\mathsf{T} \neq \mathsf{T}'$ *are* strongly swappable*, in symbols* $\mathsf{T} \# \mathsf{T}'$*, when there exist* $W, W', R, R' \subseteq \mathbb{P}$ *such that* $W \models^w \mathsf{T}$*,* $W' \models^w \mathsf{T}'$*,* $R \models^r \mathsf{T}$*,* $R' \models^r \mathsf{T}'$*, and:*

$$\big(R \cup W\big) \cap W' \;=\; \emptyset \;=\; \big(R' \cup W'\big) \cap W \qquad \diamond$$

Example 5. Let $\mathtt{f}_1()\{\mathtt{skip}\}$ and $\mathtt{f}_2(x)\{x.\mathtt{transfer}(\mathtt{value})\}$ be functions of the contracts $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively, and consider the following transactions:

$$\mathsf{T}_1 = \mathcal{A} \xrightarrow{1} \mathcal{C}_1 : \mathtt{f}_1() \qquad\qquad \mathsf{T}_2 = \mathcal{B} \xrightarrow{1} \mathcal{C}_2 : \mathtt{f}_2(\mathcal{F})$$

where $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{F}$ are account addresses. To prove that $\mathsf{T}_1 \# \mathsf{T}_2$, consider the following safe approximations of the written/read keys of T_1 and T_2, respectively:

$$W_1 = \{\mathcal{A}.\mathtt{balance}, \mathcal{C}_1.\mathtt{balance}\} \models^w \mathsf{T}_1 \qquad R_1 = \{\mathcal{A}.\mathtt{balance}\} \models^r \mathsf{T}_1$$
$$W_2 = \{\mathcal{B}.\mathtt{balance}, \mathcal{F}.\mathtt{balance}\} \models^w \mathsf{T}_2 \qquad R_2 = \{\mathcal{B}.\mathtt{balance}\} \models^r \mathsf{T}_2$$

Since $(W_1 \cup R_1) \cap W_2 = \emptyset = (W_2 \cup R_2) \cap W_1$, the two transactions are strongly swappable. Now, let:

$$\mathsf{T}_3 = \mathcal{B} \xrightarrow{1} \mathcal{C}_2 : \mathtt{f}_2(\mathcal{A})$$

and consider the following safe approximations W_3 and R_3:

$$W_3 = \{\mathcal{B}.\mathtt{balance}, \mathcal{A}.\mathtt{balance}\} \models^w \mathsf{T}_3 \qquad R_3 = \{\mathcal{B}.\mathtt{balance}\} \models^r \mathsf{T}_3$$

Since $W_1 \cap W_3 \neq \emptyset \neq W_2 \cap W_3$, then $\neg(\mathsf{T}_1 \# \mathsf{T}_3)$ and $\neg(\mathsf{T}_2 \# \mathsf{T}_3)$. $\diamond$

The following theorem ensures the soundness of our approximation, i.e. that if two transactions are strongly swappable, then they are also swappable. The converse implication does not hold, as witnessed by Example 6.

Theorem 2. $\mathsf{T} \# \mathsf{T}' \implies \mathsf{T} \rightleftarrows \mathsf{T}'$. $\diamond$

Example 6 (Swappable transactions, not strongly). Consider the following functions and transactions of a contract at address $\mathcal{C}$:

$$\mathtt{f}_1()\{\mathtt{if}\ \mathtt{sender} = \mathcal{A}\ \&\&\ k_1 = 0\ \mathtt{then}\ k_1{:=}1\ \mathtt{else}\ \mathtt{throw}\} \qquad \mathsf{T}_1 = \mathcal{A} \xrightarrow{1} \mathcal{C} : \mathtt{f}_1()$$
$$\mathtt{f}_2()\{\mathtt{if}\ \mathtt{sender} = \mathcal{B}\ \&\&\ k_2 = 0\ \mathtt{then}\ k_2{:=}1\ \mathtt{else}\ \mathtt{throw}\} \qquad \mathsf{T}_2 = \mathcal{B} \xrightarrow{1} \mathcal{C} : \mathtt{f}_2()$$

We prove that $\mathsf{T}_1 \rightleftarrows \mathsf{T}_2$. First, consider a state σ such that $\sigma\mathcal{A}\mathtt{balance} > 1$, $\sigma\mathcal{B}\mathtt{balance} > 1$, $\sigma\mathcal{C}\mathtt{balance} = n$, $\sigma\mathcal{C}k_1 = 0$ and $\sigma\mathcal{C}k_2 = 0$. We have that:

$$[\![\mathsf{T}_1\mathsf{T}_2]\!]_\sigma = \sigma\{{}^1\!/_{\mathcal{C}.k_1}, {}^1\!/_{\mathcal{C}.k_2}, {}^{n+2}\!/_{\mathcal{C}.\mathtt{balance}}\} = [\![\mathsf{T}_2\mathsf{T}_1]\!]_\sigma$$

In the second case, let σ be such that $\sigma\mathcal{A}\texttt{balance} < 1$, or $\sigma\mathcal{B}\texttt{balance} < 1$, or $\sigma\mathcal{C}k_1 \neq 0$, or $\sigma\mathcal{C}k_2 \neq 0$. It is not possible that the guards in $\texttt{f}_1$ and $\texttt{f}_2$ are both true, so T_1 or T_2 raise an exception, leaving the state unaffected. Then, also in this case we have that $[\![\mathsf{T}_1\mathsf{T}_2]\!]_\sigma = [\![\mathsf{T}_2\mathsf{T}_1]\!]_\sigma$, and so T_1 and T_2 are swappable. However, they are *not* strongly swappable if there exist reachable states σ, σ' such that $\sigma\mathcal{C}k_1 = 0 = \sigma'\mathcal{C}k_2$. To see why, let $W_1 = \{\mathcal{A}.\texttt{balance}, \mathcal{C}.\texttt{balance}, \mathcal{C}.k_1\}$. From the code of $\texttt{f}_0$ we see that W_1 is the least safe over-approximation of the written keys of T_1 ($W_1 \models^w \mathsf{T}_1$). This means that every safe approximation of T_1 must include the keys of W_1. Similarly, $W_2 = \{\mathcal{B}.\texttt{balance}, \mathcal{C}.\texttt{balance}, \mathcal{C}.k_2\}$ is the least safe over-approximation of the written keys of T_2 ($W_2 \models^w \mathsf{T}_2$). Since the least safe approximations of the keys written by T_1 and T_2 are not disjoint, $\mathsf{T}_1 \# \mathsf{T}_2$ does not hold. ◇

Theorem 3 states that the Mazurkiewicz equivalence $\simeq_\#$ is stricter than $\simeq_\rightleftarrows$. Together with Theorem 1, if $\mathbf{B}$ is transformed into $\mathbf{B'}$ by exchanging adjacent strongly swappable transactions, then $\mathbf{B}$ and $\mathbf{B'}$ are observationally equivalent.

Theorem 3. $\simeq_\# \subseteq \simeq_\rightleftarrows$. ◇

Note that if the contract language is Turing-equivalent, then finding approximations which satisfy the disjointness condition in Definition 6 is not computable, and so the relation $\#$ is undecidable.

Parameterised strong swappability Strongly swappability abstracts from the actual static analysis used to obtain the safe approximations: it is sufficient that such an analysis exists. Definition 7 below parameterises strong swappability over a static analysis, which we represent as a function from transactions to sets of qualified keys, just requiring it to be a safe approximation. Formally, we say that W is a *static analysis of written keys* when $W(\mathsf{T}) \models^w \mathsf{T}$, for all T; similarly, R is a *static analysis of read keys* when $R(\mathsf{T}) \models^r \mathsf{T}$, for all T.

Definition 7 (Parameterised strong swappability). *Let W and R be static analyses of written/read keys. We say that T, T' are* strongly swappable w.r.t. *W and R, in symbols $\mathsf{T} \#^W_R \mathsf{T}'$, if:*

$$\big(R(\mathsf{T}) \cup W(\mathsf{T})\big) \cap W(\mathsf{T}') \;=\; \emptyset \;=\; \big(R(\mathsf{T}') \cup W(\mathsf{T}')\big) \cap W(\mathsf{T})$$ ◇

Note that an effective procedure for computing W and R gives an effective procedure to determine whether two transactions are (strongly) swappable.

Lemma 3. *For all static analyses W and R: (i) $\#^W_R \subseteq \#$; (ii) if W and R are computable, then $\#^W_R$ is decidable.* ◇

From the inclusion in item (i) of Lemma 3 and from Theorem 3 we obtain:

Theorem 4. $\simeq_{\#^W_R} \subseteq \simeq_\# \subseteq \simeq_\rightleftarrows$. ◇

4 True concurrency for blockchains

Given a swappability relation $\mathcal{R}$, we transform a sequence of transactions $\mathbf{B}$ into an *occurrence net* $\mathsf{N}_{\mathcal{R}}(\mathbf{B})$, which describes the partial order induced by $\mathcal{R}$. Any concurrent execution of the transactions in $\mathbf{B}$ which respects this partial order is equivalent to the serial execution of $\mathbf{B}$ (Theorem 5).

From blockchains to occurrence nets We start by recapping the notion of Petri net [19]. A *Petri net* is a tuple $\mathsf{N} = (\mathsf{P}, \mathsf{Tr}, \mathsf{F}, \mathsf{m}_0)$, where P is a set of *places*, Tr is a set of *transitions* (with $\mathsf{P} \cap \mathsf{Tr} = \emptyset$), and $\mathsf{F} : (\mathsf{P} \times \mathsf{Tr}) \cup (\mathsf{Tr} \times \mathsf{P}) \to \mathbb{N}$ is a *weight function*. The state of a net is a *marking*, i.e. a multiset $\mathsf{m} : \mathsf{P} \to \mathbb{N}$ defining how many *tokens* are contained in each place; we denote with m_0 the initial marking. The behaviour of a Petri net is specified as a transition relation between markings: intuitively, a transition t is enabled at m when each place p has at least $\mathsf{F}(\mathsf{p},\mathsf{t})$ tokens in m. When an enabled transition t is fired, it consumes $\mathsf{F}(\mathsf{p},\mathsf{t})$ tokens from each p, and produces $\mathsf{F}(\mathsf{t},\mathsf{p}')$ tokens in each p'. Formally, given $x \in \mathsf{P} \cup \mathsf{Tr}$, we define the *preset* ${}^\bullet x$ and the *postset* $x^\bullet$ as multisets: ${}^\bullet x(y) = \mathsf{F}(y,x)$, and $x^\bullet(y) = \mathsf{F}(x,y)$. A transition t is *enabled* at m when ${}^\bullet\mathsf{t} \subseteq \mathsf{m}$. The transition relation between markings is defined as $\mathsf{m} \xrightarrow{\mathsf{t}} \mathsf{m} - {}^\bullet\mathsf{t} + \mathsf{t}^\bullet$, where t is enabled. We say that $\mathsf{t}_1 \cdots \mathsf{t}_n$ is a *firing sequence from* m *to* m' when $\mathsf{m} \xrightarrow{\mathsf{t}_1} \cdots \xrightarrow{\mathsf{t}_n} \mathsf{m}'$, and in this case we say that m' is *reachable from* m. We say that m' is *reachable* when it is reachable from m_0.

An *occurrence net* [8] is a Petri net such that: (i) $|\mathsf{p}^\bullet| \leq 1$ for all p; (ii) $|{}^\bullet\mathsf{p}| = 1$ if $\mathsf{p} \notin \mathsf{m}_0$, and $|{}^\bullet\mathsf{p}| = 0$ if $\mathsf{p} \in \mathsf{m}_0$; (iii) F is a relation, i.e. $\mathsf{F}(x,y) \leq 1$ for all x, y; (iv) F^* is a acyclic, i.e. $\forall x, y \in \mathsf{P} \cup \mathsf{Tr} : (x,y) \in \mathsf{F}^* \wedge (y,x) \in \mathsf{F}^* \implies x = y$ (where F^* is the reflexive and transitive closure of F).

In Figure 2 we transform a blockchain $\mathbf{B} = \mathsf{T}_1 \cdots \mathsf{T}_n$ into a Petri net $\mathsf{N}_{\mathcal{R}}(\mathbf{B})$, where $\mathcal{R}$ is an arbitrary relation between transactions. Although any relation $\mathcal{R}$ ensures that $\mathsf{N}_{\mathcal{R}}(\mathbf{B})$ is an occurrence net (Lemma 4 below), our main results hold when $\mathcal{R}$ is a strong swappability relation. The transformation works as follows: the i-th transaction in $\mathbf{B}$ is rendered as a transition (T_i, i) in $\mathsf{N}_{\mathcal{R}}(\mathbf{B})$, and transactions related by $\mathcal{R}$ are transformed into concurrent transitions. Technically, this concurrency is specified as a relation $<$ between transitions, such that $(\mathsf{T}_i, i) < (\mathsf{T}_j, j)$ whenever $i < j$, but T_i and T_j are not related by $\mathcal{R}$. The places, the weight function, and the initial marking of $\mathsf{N}_{\mathcal{R}}(\mathbf{B})$ are chosen to ensure that the firing ot transitions respects the relation $<$.

Example 7. Consider the following transactions and functions of a contract $\mathcal{C}$:

$$\mathsf{T}_{\mathtt{f}} = \mathcal{A} \xrightarrow{0} \mathcal{C} : \mathtt{f}() \qquad \mathtt{f}()\,\{\mathtt{if}\ x = 0\ \mathtt{then}\ y{:=}1\ \mathtt{else}\ \mathtt{throw}\}$$

$$\mathsf{T}_{\mathtt{g}} = \mathcal{A} \xrightarrow{0} \mathcal{C} : \mathtt{g}() \qquad \mathtt{g}()\,\{\mathtt{if}\ y = 0\ \mathtt{then}\ x{:=}1\ \mathtt{else}\ \mathtt{throw}\}$$

$$\mathsf{T}_{\mathtt{h}} = \mathcal{A} \xrightarrow{0} \mathcal{C} : \mathtt{h}() \qquad \mathtt{h}()\,\{z{:=}1\}$$

Let $P^w_{\mathtt{f}} = P^r_{\mathtt{g}} = \{\mathcal{C}.y\}, P^r_{\mathtt{f}} = P^w_{\mathtt{g}} = \{\mathcal{C}.x\}, P^w_{\mathtt{h}} = \{\mathcal{C}.z\}, P^r_{\mathtt{h}} = \emptyset$. It is easy to check that these sets are safe approximations of their transactions (e.g., $P^w_{\mathtt{f}}$ safely

$$\mathsf{Tr} = \{(\mathsf{T}_i, i) \mid 1 \leq i \leq n\}$$

$$\mathsf{P} = \{(*, \mathtt{t}) \mid \mathtt{t} \in \mathsf{Tr}\} \cup \{(\mathtt{t}, *) \mid \mathtt{t} \in \mathsf{Tr}\} \cup \{(\mathtt{t}, \mathtt{t}') \mid \mathtt{t} < \mathtt{t}'\}$$
$$\text{where } (\mathsf{T}, i) < (\mathsf{T}', j) \triangleq (i < j) \wedge \neg(\mathsf{T} \mathcal{R} \mathsf{T}')$$

$$\mathsf{F}(x,y) = \begin{cases} 1 & \text{if } y = \mathtt{t} \text{ and } \big(x = (*, \mathtt{t}) \text{ or } x = (\mathtt{t}', \mathtt{t})\big) \\ 1 & \text{if } x = \mathtt{t} \text{ and } \big(y = (\mathtt{t}, *) \text{ or } y = (\mathtt{t}, \mathtt{t}')\big) \\ 0 & \text{otherwise} \end{cases} \qquad \mathsf{m}_0(\mathsf{p}) = \begin{cases} 1 & \text{if } \mathsf{p} = (*, \mathtt{t}) \\ 0 & \text{otherwise} \end{cases}$$

Fig. 2: Construction of a Petri net from a blockchain $\mathbf{B} = \mathsf{T}_1 \cdots \mathsf{T}_n$.

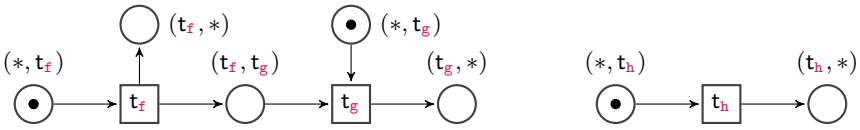

Fig. 3: Occurrence net for Example 7.

approximates the keys written by $\mathsf{T}_\mathtt{f}$). By Definition 6 we have that $\mathsf{T}_\mathtt{f} \# \mathsf{T}_\mathtt{h}$, $\mathsf{T}_\mathtt{g} \# \mathsf{T}_\mathtt{h}$, but $\neg(\mathsf{T}_\mathtt{f} \# \mathsf{T}_\mathtt{g})$. We display $\mathsf{N}_\#(\mathsf{T}_\mathtt{f}\,\mathsf{T}_\mathtt{h}\,\mathsf{T}_\mathtt{g})$ in Figure 3, where $\mathtt{t}_\mathtt{f} = (\mathsf{T}_\mathtt{f}, 1)$, $\mathtt{t}_\mathtt{h} = (\mathsf{T}_\mathtt{h}, 2)$, and $\mathtt{t}_\mathtt{g} = (\mathsf{T}_\mathtt{g}, 3)$. Note that $\mathtt{t}_\mathtt{g}$ can only be fired after $\mathtt{t}_\mathtt{f}$, while $\mathtt{t}_\mathtt{h}$ can be fired independently from $\mathtt{t}_\mathtt{f}$ and $\mathtt{t}_\mathtt{g}$. This is coherent with the fact that $\mathsf{T}_\mathtt{h}$ is swappable with both $\mathsf{T}_\mathtt{f}$ and $\mathsf{T}_\mathtt{g}$, while $\mathsf{T}_\mathtt{f}$ and $\mathsf{T}_\mathtt{g}$ are not swappable. ◇

Lemma 4. *$\mathsf{N}_\mathcal{R}(\mathbf{B})$ is an occurrence net, for all $\mathcal{R}$ and $\mathbf{B}$.*

Step firing sequences Theorem 5 below establishes a correspondence between concurrent and serial execution of transactions. Since the semantics of serial executions is given in terms of blockchain states σ, to formalise this correspondence we use the same semantics domain also for concurrent executions. This is obtained in two steps. First, we define concurrent executions of $\mathbf{B}$ as the *step firing sequences* (i.e. finite sequences of *sets* of transitions) of $\mathsf{N}_\#(\mathbf{B})$. Then, we give a semantics to step firing sequences, in terms of blockchain states.

We denote finite sets of transitions, called *steps*, as $\mathsf{U}, \mathsf{U}', \ldots$. Their preset and postset are defined as ${}^\bullet\mathsf{U} = \sum_{\mathsf{p} \in \mathsf{U}} {}^\bullet\mathsf{p}$ and $\mathsf{U}^\bullet = \sum_{\mathsf{p} \in \mathsf{U}} \mathsf{p}^\bullet$, respectively. We say that U is *enabled at* m when ${}^\bullet\mathsf{U} \leq \mathsf{m}$, and in this case firing U results in the move $\mathsf{m} \xrightarrow{\mathsf{U}} \mathsf{m} - {}^\bullet\mathsf{U} + \mathsf{U}^\bullet$. Let $\mathbf{U} = \mathsf{U}_1 \cdots \mathsf{U}_n$ be a finite sequence of steps. We say that $\mathbf{U}$ is a *step firing sequence from* m *to* m' if $\mathsf{m} \xrightarrow{\mathsf{U}_1} \cdots \xrightarrow{\mathsf{U}_n} \mathsf{m}'$, and in this case we write $\mathsf{m} \xrightarrow{\mathbf{U}} \mathsf{m}'$.

Concurrent execution of transactions We now define how to execute transactions in parallel. The idea is to execute transactions in *isolation*, and then merge their changes, whenever they are mutually disjoint. The state updated resulting from the execution of a transaction are formalised as in Definition 1.

An *update collector* is a function Π that, given a state σ and a transaction T, gives an update $\pi = \Pi(\sigma, \mathsf{T})$ which maps (at least) the updated qualified keys to their new values. In practice, update collectors can be obtained by instrumenting the run-time environment of smart contracts, so to record the state changes resulting from the execution of transactions. We formalise update collectors abstracting from the implementation details of such an instrumentation:

Definition 8 (Update collector). *We say that a function Π is an* update collector *when $[\![\mathsf{T}]\!]_\sigma = \sigma(\Pi(\sigma, \mathsf{T}))$, for all σ and T.* ◇

There exists a natural ordering of collectors, which extends the ordering between state updates (i.e., set inclusion, when interpreting them as sets of substitutions): namely, $\Pi \sqsubseteq \Pi'$ holds when $\forall \sigma, \mathsf{T} : \Pi(\sigma, \mathsf{T}) \subseteq \Pi'(\sigma, \mathsf{T})$. The following lemma characterizes the least update collector w.r.t. this ordering.

Lemma 5 (Least update collector). *Let $\Pi^\star(\sigma, \mathsf{T}) = [\![\mathsf{T}]\!]_\sigma - \sigma$, where we define $\sigma' - \sigma$ as $\bigcup_{\sigma' p \neq \sigma p} \{{}^{\sigma' p}/_{p}\}$. Then, $\Pi^\star$ is the least update collector.* ◇

The merge of two state updates is the union of the corresponding substitutions; to avoid collisions, we make the merge operator undefined when the domains of the two updates overlap.

Definition 9 (Merge of state updates). *Let π_0, π_1 be state updates. When $\mathrm{keys}(\pi_0) \cap \mathrm{keys}(\pi_1) = \emptyset$, we define $\pi_0 \oplus \pi_1$ as follows:*

$$(\pi_0 \oplus \pi_1)p = \begin{cases} \pi_0 p & \text{if } p \in \mathrm{keys}(\pi_0) \\ \pi_1 p & \text{if } p \in \mathrm{keys}(\pi_1) \\ \bot & \text{otherwise} \end{cases}$$ ◇

The merge operator enjoys the commutative monoidal laws, and can therefore be extended to (finite) sets of state updates.

We now associate step firing sequences with state updates. The semantics of a step $\mathsf{U} = \{(\mathsf{T}_1, 1), \ldots, (\mathsf{T}_n, n)\}$ in σ is obtained by applying to σ the merge of the updates $\Pi(\sigma, \mathsf{T}_i)$, for all $i \in 1..n$ — whenever the merge is defined. The semantics of a step firing sequence is then obtained by folding that of its steps.

Definition 10 (Semantics of step firing sequences). *We define the semantics of step firing sequences, given Π and σ, as:*

$$[\![\epsilon]\!]^\Pi_\sigma = \sigma \qquad [\![\mathsf{U}\mathbf{U}]\!]^\Pi_\sigma = [\![\mathbf{U}]\!]^\Pi_{\sigma'} \quad \text{where } \sigma' = [\![\mathsf{U}]\!]^\Pi_\sigma = \sigma \bigoplus_{(\mathsf{T},i)\in\mathsf{U}} \Pi(\sigma, \mathsf{T})$$ ◇

Example 8. Let $\mathtt{t_f}$, $\mathtt{t_g}$, and $\mathtt{t_h}$ be as in Example 7, and let $\sigma\mathtt{c}x = \sigma\mathtt{c}y = 0$. Since $\Pi^\star(\sigma, \mathsf{T_f}) = \{{}^1/_{\mathtt{c}.y}\}$, $\Pi^\star(\sigma, \mathsf{T_g}) = \{{}^1/_{\mathtt{c}.x}\}$, and $\Pi^\star(\sigma, \mathsf{T_h}) = \{{}^1/_{\mathtt{c}.z}\}$, we have:

$$[\![\{\mathtt{t_f}, \mathtt{t_h}\}]\!]^{\Pi^\star}_\sigma = \sigma(\{{}^1/_{\mathtt{c}.y}\} \oplus \{{}^1/_{\mathtt{c}.z}\}) = \sigma\{{}^1/_{\mathtt{c}.y}, {}^1/_{\mathtt{c}.z}\}$$
$$[\![\{\mathtt{t_g}, \mathtt{t_h}\}]\!]^{\Pi^\star}_\sigma = \sigma(\{{}^1/_{\mathtt{c}.x}\} \oplus \{{}^1/_{\mathtt{c}.z}\}) = \sigma\{{}^1/_{\mathtt{c}.x}, {}^1/_{\mathtt{c}.z}\}$$
$$[\![\{\mathtt{t_f}, \mathtt{t_g}\}]\!]^{\Pi^\star}_\sigma = (\sigma\{{}^1/_{\mathtt{c}.y}\} \oplus \{{}^1/_{\mathtt{c}.x}\}) = \sigma\{{}^1/_{\mathtt{c}.y}, {}^1/_{\mathtt{c}.x}\}$$

Note that, for all σ:

$$\begin{array}{rcl} [\![\mathsf{T_f}\,\mathsf{T_h}]\!]_\sigma & = & [\![\mathsf{T_h}\,\mathsf{T_f}]\!]_\sigma \;=\; \sigma\{1/\mathtt{e}.y, 1/\mathtt{e}.z\} \;=\; [\![\{\mathsf{t_f}, \mathsf{t_h}\}]\!]^{\Pi^\star}_\sigma \\ [\![\mathsf{T_g}\,\mathsf{T_h}]\!]_\sigma & = & [\![\mathsf{T_h}\,\mathsf{T_g}]\!]_\sigma \;=\; \sigma\{1/\mathtt{e}.x, 1/\mathtt{e}.z\} \;=\; [\![\{\mathsf{t_g}, \mathsf{t_h}\}]\!]^{\Pi^\star}_\sigma \end{array}$$

So, the serial execution of $\mathsf{T_f}$ and $\mathsf{T_h}$ (in both orders) is equal to their concurrent execution (similarly for $\mathsf{T_g}$ and $\mathsf{T_h}$). Instead, for all σ such that $\sigma\mathtt{e}x = \sigma\mathtt{e}y = 0$:

$$[\![\mathsf{T_f}\,\mathsf{T_g}]\!]_\sigma = \sigma\{1/\mathtt{e}.y\} \qquad [\![\mathsf{T_g}\,\mathsf{T_f}]\!]_\sigma = \sigma\{1/\mathtt{e}.x\} \qquad [\![\{\mathsf{t_f}, \mathsf{t_g}\}]\!]^{\Pi^\star}_\sigma = \sigma\{1/\mathtt{e}.y, 1/\mathtt{e}.x\}$$

So, concurrent executions of $\mathsf{T_f}$ and $\mathsf{T_g}$ may differ from serial ones. This is coherent with the fact that, in Figure 3, $\mathsf{t_f}$ and $\mathsf{t_g}$ are *not* concurrent. ◇

Concurrent execution of blockchains Theorem 5 relates serial executions of transactions to concurrent ones (which are rendered as step firing sequences). Item (a) establishes a confluence property: if two step firing sequences lead to the same marking, then they also lead to the same blockchain state. Item (b) ensures that the blockchain, interpreted as a sequence of transitions, is a step firing sequence, and it is *maximal* (i.e., there is a bijection between the transactions in the blockchain and the transitions of the corresponding net). Finally, item (c) ensures that executing maximal step firing sequences is equivalent to executing serially the blockchain.

Theorem 5. *Let* $\mathsf{B} = \mathsf{T}_1 \cdots \mathsf{T}_n$. *Then, in* $\mathsf{N}_{\#}(\mathsf{B})$:

(a) if $\mathsf{m}_0 \xrightarrow{\mathsf{U}} \mathsf{m}$ *and* $\mathsf{m}_0 \xrightarrow{\mathsf{U'}} \mathsf{m}$, *then* $[\![\mathsf{U}]\!]^{\Pi^\star}_\sigma = [\![\mathsf{U'}]\!]^{\Pi^\star}_\sigma$, *for all reachable* σ;
(b) $\{(\mathsf{T}_1, 1)\} \cdots \{(\mathsf{T}_n, n)\}$ *is a maximal step firing sequence;*
(c) for all maximal step firing sequences U, *for all reachable* σ, $[\![\mathsf{U}]\!]^{\Pi^\star}_\sigma = [\![\mathsf{B}]\!]_\sigma$.

Remarkably, the implications of Theorem 5 also apply to $\mathsf{N}_{\#^W_R}(\mathsf{B})$.

Example 9. Recall $\mathsf{B} = \mathsf{T_f}\,\mathsf{T_h}\,\mathsf{T_g}$ and $\mathsf{N}_{\#}(\mathsf{B})$ from Example 7, let $\mathsf{U} = \{\mathsf{t_f}, \mathsf{t_h}\}\{\mathsf{t_g}\}$, and let σ be such that $\sigma\mathtt{e}x = \sigma\mathtt{e}y = 0$. As predicted by item (c) of Theorem 5:

$$[\![\mathsf{B}]\!]_\sigma \;=\; \sigma\{1/\mathtt{e}.y\}\{1/\mathtt{e}.z\} \;=\; [\![\mathsf{U}]\!]^{\Pi^\star}_\sigma$$

Let $\mathsf{U'} = \{\mathsf{t_f}\}\{\mathsf{t_g}, \mathsf{t_h}\}$. We have that U and $\mathsf{U'}$ lead to the same marking, where the places $(\mathsf{t_f}, *)$, $(\mathsf{t_g}, *)$ and $(\mathsf{t_h}, *)$ contain one token each, while the other places have no tokens. By item (a) of Theorem 5 we conclude that $[\![\mathsf{U}]\!]^{\Pi^\star}_\sigma = [\![\mathsf{U'}]\!]^{\Pi^\star}_\sigma$. Now, let $\mathsf{U''} = \{\mathsf{t_h}\}\{\mathsf{t_f}, \mathsf{t_g}\}$. Note that, although $\mathsf{U''}$ is maximal, it is not a step firing sequence, since the second step is not enabled (actually, $\mathsf{t_f}$ and $\mathsf{t_g}$ are not concurrent, as pointed out in Example 8). Therefore, the items of Theorem 5 do not apply to $\mathsf{U''}$, coherently with the fact that $\mathsf{U''}$ does not represent any sequential execution of B. ◇

5 Case study: ERC-721 token

We now apply our theory to an archetypal Ethereum smart contract, which implements a "non-fungible token" following the standard ERC-721 interface [14,15]. This contract defines the functions to transfer tokens between users, and to delegate their trade to other users. Currently, token transfers involve $\sim 50\%$ of the transactions on the Ethereum blockchain [1], with larger peaks due to popular contracts like Cryptokitties [22].

We sketch below the implementation of the `Token` contract, using Solidity, the main high-level smart contract language in Ethereum (see Appendix B of [6] for the full implementation).

The contract state is defined by the following mappings:

```
mapping(uint256 => address) owner;
mapping(uint256 => bool) exists;
mapping(address => uint256) balance;
mapping (address => mapping (address => bool)) operatorApprovals;
```

Each token is uniquely identified by an integer value (of type `uint256`), while users are identified by an `address`. The mapping `owner` maps tokens to their owners' addresses (the zero address is used to denote a dummy owner). The mapping `exists` tells whether a token has been created or not, while `balance` gives the number of tokens owned by each user. The mapping `operatorApprovals` allows a user to delegate the transfer of all her tokens to third parties.

The function `transferFrom` transfers a token from the owner to another user. The `require` assertion rules out some undesirable cases, e.g., if the token does not exist, or it is not owned by the `from` user, or the user attempts to transfer the token to himself. Once all these checks are passed, the transfer succeeds if the `sender` of the transaction owns the token, or if he has been delegated by the owner. The mappings `owner` and `balance` are updated as expected.

```
function transferFrom(address from, address to, uint256 id)
    external {
  require (exists[id] && from==owner[id]
          && from!=to && to!=address(0));
  if (from==msg.sender || operatorApprovals[from][msg.sender]) {
    owner[id] = to;
    balance[from] -= 1;
    balance[to] += 1;
  }
}
```

The function `setApprovalForAll` delegates the transfers of all the tokens of the `sender` to the `operator` when the boolean `isApproved` is true, otherwise it revokes the delegation.

```
function setApprovalForAll(address operator, bool isApproved)
    external {
  operatorApprovals[msg.sender][operator] = isApproved;
}
```

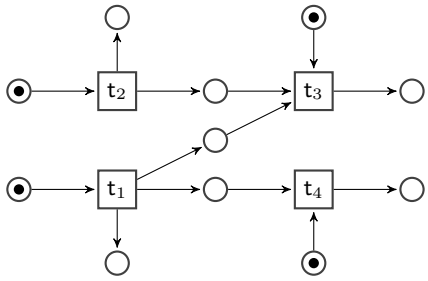

Fig. 4: Occurrence net for the blockchain $\mathbf{B} = \mathsf{T}_1\mathsf{T}_2\mathsf{T}_3\mathsf{T}_4$ of the ERC-721 token.

Assume that user $\mathcal{A}$ owns two tokens, identified by the integers 1 and 2, and consider the following transactions:

$$\mathsf{T}_1 = \mathcal{A} \xrightarrow{0} \texttt{Token} : \texttt{transferFrom}(\mathcal{A}, \mathcal{P}, 1)$$

$$\mathsf{T}_2 = \mathcal{A} \xrightarrow{0} \texttt{Token} : \texttt{setApprovalForAll}(\mathcal{B}, \texttt{true})$$

$$\mathsf{T}_3 = \mathcal{B} \xrightarrow{0} \texttt{Token} : \texttt{transferFrom}(\mathcal{A}, \mathcal{Q}, 2)$$

$$\mathsf{T}_4 = \mathcal{P} \xrightarrow{0} \texttt{Token} : \texttt{transferFrom}(\mathcal{P}, \mathcal{B}, 1)$$

We have that $\mathsf{T}_1 \# \mathsf{T}_2$, $\mathsf{T}_2 \# \mathsf{T}_4$, and $\mathsf{T}_3 \# \mathsf{T}_4$ (this can be proved e.g. by using the static approximations in Appendix B of [6]), while the other combinations are not swappable. Let $\mathbf{B} = \mathsf{T}_1\mathsf{T}_2\mathsf{T}_3\mathsf{T}_4$. The resulting occurrence net is displayed in Figure 4. For instance, let $\mathbf{U} = \{\mathsf{T}_1, \mathsf{T}_2\}\{\mathsf{T}_3, \mathsf{T}_4\}$, i.e. T_1 and T_2 are executed concurrently, as well as T_3 and T_4. From item (c) of Theorem 5 we have that this concurrent execution is equivalent to the serial one.

Although this example deals with the marginal case where the sender and the receiver of tokens overlap, in practice the large majority of transactions in a block either involves distinct users, or invokes distinct ERC-721 interfaces, making it possible to increase the degree of concurrency of `transferFrom` transactions.

6 Related work and conclusions

We have proposed a static approach to improve the performance of blockchains by concurrently executing transactions. We have started by introducing a model of transactions and blockchains. We have defined two transactions to be *swappable* when inverting their order does not affect the blockchain state. We have then introduced a static approximation of swappability, based on a static analysis of the sets of keys read/written by transactions. We have rendered concurrent executions of a sequence of transactions as *step firing sequences* in the associated occurrence net. Our main technical result, Theorem 5, shows that these concurrent executions are semantically equivalent to the sequential one.

We can exploit our results in practice to improve the performances of miners and validators. Miners should perform the following steps to mine a block:

1. gather from the network a set of transactions, and put them in an arbitrary linear order $\mathbf{B}$, which is the mined block;
2. compute the relation $\#_R^W$ on $\mathbf{B}$, using a static analysis of read/written keys;
3. construct the occurrence net $\mathsf{N}_{\#_R^W}(\mathbf{B})$;
4. execute transactions concurrently according to the occurrence net, exploiting the available parallelism.

The behaviour of validators is almost identical to that of miners, except that in step (1), rather than choosing the order of transactions, they should adhere to the ordering of the mined block $\mathbf{B}$. Note that in the last step, validators can execute any maximal step firing sequence which is coherent with their degree of parallelism: item (c) of Theorem 5 ensures that the resulting state is equal to the state obtained by the miner. The experiments in [12] suggest that parallelization may lead to a significant improvement of the performance of nodes: the benchmarks on a selection of representative contracts show an overall speedup of 1.33x for miners and 1.69x for validators, using only three cores.

Note that malevolent users could attempt a denial-of-service attack by publishing contracts which are hard to statically analyse, and therefore are not suitable for parallelization. This kind of attacks can be mitigated by adopting a mining strategy that gives higher priority to parallelizable transactions.

Applying our approach to Ethereum Applying our theory to Ethereum would require a static analysis of read/written keys at the level of EVM bytecode. As far as we know, the only tool implementing such an analysis is ES-ETH [16]. However, the current version of the tool has several limitations, like e.g. the compile-time approximation of dictionary keys and of values shorter than 32 bytes, which make ES-ETH not directly usable to the purposes of our work. In general, precise static analyses at the level of the Ethereum bytecode are difficult to achieve, since the language has features like dynamic dispatching and pointer aliasing which are notoriously a source of imprecision for static analysis. However, coarser approximations of read/written keys may be enough to speed-up the execution of transactions. For instance, in Ethereum, blocks typically contain many transactions which transfer tokens between participants, and many of them involve distinct senders and receivers. A relatively simple analysis of the code of token contracts (which is usually similar to that in Section 5) may be enough to detect that these transactions are swappable.

Aiming at minimality, our model does not include the *gas mechanism*, which is used in Ethereum to pay miners for executing contracts. The sender of a transaction deposits into it some crypto-currency, to be paid to the miner which appends the transaction to the blockchain. Each instruction executed by the miner consumes part of this deposit; when the deposit reaches zero, the miner stops executing the transaction. At this point, all the effects of the transaction (except the payment to the miner) are rolled back. Our transaction model could be easily extended with a gas mechanism, by associating a cost to statements and recording the gas consumption in the environment. Remarkably, adding gas does not invalidate approximations of read/written keys which are correct while

neglecting gas. However, a gas-aware analysis may be more precise of a gas-oblivious one: for instance, in the statement $\texttt{if}\ k\ \texttt{then}\ \texttt{f}_{long}(); x{:=}1\ \texttt{else}\ y{:=}1$ (where $\texttt{f}_{long}$ is a function which exceeds the available gas) a gas-aware analysis would be able to detect that x is not written.

Related work A few works study how to optimize the execution of smart contracts on Ethereum, using dynamic techniques adopted from software transactional memory [4,12,13]. These works are focussed on empirical aspects (e.g., measuring the speedup obtained on a given benchmark), while we focus on the theoretical counterpart. In [12,13], miners execute a set of transactions speculatively in parallel, using abstract locks and inverse logs to dynamically discover conflicts and to recover from inconsistent states. The obtained execution is guaranteed to be equivalent to a serial execution of the same set of transactions. The work [4] proposes a conceptually similar technique, but based on optimistic software transactional memory. Since speculative execution is non-deterministic, in both approaches miners need to communicate the chosen schedule of transactions to validators, to allow them to correctly validate the block. This schedule must be embedded in the mined block: since Ethereum does not support this kind of block metadata, these approaches would require a "soft-fork" of the blockchain to be implemented in practice. Instead, our approach is compatible with the current Ethereum, since miners only need to append transactions to the blockchain. Compared to [12,4], where conflicts are detected dynamically, our approach relies on a static analysis to detect potential conflicts. Since software transactional memory introduces a run-time overhead, in principle a static technique could allow for faster executions, at the price of a preprocessing phase. Saraph and Herlihy [20] study the effectiveness of speculatively executing smart contracts in Ethereum. They sample past blocks of transactions (from July 2016 to December 2017), replay them by using a speculative execution engine, and measure the speedup obtained by parallel execution. Their results show that simple speculative strategies yield non-trivial speed-ups. Further, they note that many of the data conflicts (i.e. concurrent read/write accesses to the same state location) arise in periods of high traffic, and they are caused by a small number of popular contracts, like e.g. tokens.

In the permissioned setting, Hyperledger Fabric [3] follows the "execute first and then order" paradigm: transactions are executed speculatively, and then their ordering is checked for correctness [2]. In this paradigm, appending a transaction requires a few steps. First, a client proposes a transaction to a set of "endorsing" peers, which simulate the transaction without updating the blockchain. The output of the simulation includes the state updates of the transaction execution, and the sets of read/written keys. These sets are then signed by the endorsing peers, and returned to the client, which submits them to the "ordering" peers. These nodes order transactions in blocks, and send them to the "committing" peers, which validate them. A block $\mathsf{T}_1 \cdots \mathsf{T}_n$ is valid when, if a key k is read by transaction T_i, then k has not been written by a transaction T_j with $j < i$. Finally, validated blocks are appended to the blockchain. Our model is coherent with Ethereum, which does not support speculative execution.

Future works A relevant line of research is the design of domain-specific languages for smart contracts that are directly amenable to techniques that, like ours, increase the degree of concurrency of executions. For this purpose, the language should support static analyses of read/written keys, like the one we use to define the strong swappability relation. Although the literature describes various static analyses of smart contracts, most of them are focussed on finding security vulnerabilities, rather than enhancing concurrency.

Outside the realm of smart contracts, a few papers propose static analyses of read/written variables. The paper [11] describes an analysis based on separation logic, and applies it to resolve conflicts in the setting of *snapshot isolation* for transactional memory in Java. When a conflict is detected, the read/write sets are used to determine how the code can be modified to resolve it. The paper [10] presents a static analysis to infer read and write locations in a C-like language with atomic sections. The analysis is used to translate atomic sections into standard lock operations. The design of new smart contract languages could take advantage of these analyses.

Acknowledgements Massimo Bartoletti is partially supported by Aut. Reg. Sardinia projects *"Smart collaborative engineering"* and *"Sardcoin"*. Letterio Galletta is partially supported by IMT Lucca project *"PAI VeriOSS"* and by MIUR project PRIN 2017FTXR7S *"Methods and Tools for Trustworthy Smart Systems"*. Maurizio Murgia is partially supported by MIUR PON *"Distributed Ledgers for Secure Open Communities"* and by Aut. Reg. Sardinia project *"Smart collaborative engineering"*.

References

1. Ethereum token dynanics, https://stat.bloxy.info/superset/dashboard/tokens
2. Hyperledger Fabric: Read-write set semantics. https://hyperledger-fabric.readthedocs.io/en/release-1.4/readwrite.html
3. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., Caro, A.D., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolic, M., Cocco, S.W., Yellick, J.: Hyperledger Fabric: a distributed operating system for permissioned blockchains. In: EuroSys. pp. 30:1–30:15 (2018). https://doi.org/10.1145/3190508.3190538
4. Anjana, P.S., Kumari, S., Peri, S., Rathor, S., Somani, A.: An efficient framework for optimistic concurrent execution of smart contracts. In: PDP. pp. 83–92 (2019). https://doi.org/10.1109/EMPDP.2019.8671637
5. Bartoletti, M., Galletta, L., Murgia, M.: A minimal core calculus for Solidity contracts. In: Cryptocurrencies and Blockchain Technology. LNCS, vol. 11737, pp. 233–243. Springer (2019). https://doi.org/10.1007/978-3-030-31500-9_15
6. Bartoletti, M., Galletta, L., Murgia, M.: A true concurrent model of smart contracts executions. CoRR **abs/1905.04366** (2020), http://arxiv.org/abs/1905.04366
7. Bernstein, A.J.: Analysis of programs for parallel processing. IEEE Trans. on Electronic Computers **EC-15**(5), 757–763 (1966). https://doi.org/10.1109/PGEC.1966.264565

8. Best, E., Devillers, R.R.: Sequential and concurrent behaviour in petri net theory. Theor. Comput. Sci. **55**(1), 87–136 (1987). https://doi.org/10.1016/0304-3975(87)90090-9
9. Buterin, V.: Ethereum: a next generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper (2013)
10. Cherem, S., Chilimbi, T.M., Gulwani, S.: Inferring locks for atomic sections. In: ACM SIGPLAN PLDI. pp. 304–315 (2008). https://doi.org/10.1145/1375581.1375619
11. Dias, R.J., Lourenço, J.M., Preguiça, N.M.: Efficient and correct transactional memory programs combining snapshot isolation and static analysis. In: USENIX Conf. on Hot topics in Parallelism (HotPar) (2011)
12. Dickerson, T.D., Gazzillo, P., Herlihy, M., Koskinen, E.: Adding concurrency to smart contracts. In: ACM PODC. pp. 303–312 (2017). https://doi.org/10.1145/3087801.3087835
13. Dickerson, T.D., Gazzillo, P., Herlihy, M., Koskinen, E.: Adding concurrency to smart contracts. Bulletin of the EATCS **124** (2018)
14. Entriken, W., Shirley, D., Evans, J., Sachs, N.: EIP 721: ERC-721 non-fungible token standard, https://eips.ethereum.org/EIPS/eip-721
15. Fröwis, M., Fuchs, A., Böhme, R.: Detecting token systems on Ethereum. In: Financial Cryptography and Data Security. LNCS, vol. 11598, pp. 93–112. Springer (2019). https://doi.org/10.1007/978-3-030-32101-7_7
16. Marcia, D.: ES-ETH: Ethereum state change examiner. https://github.com/DiegoMarcia/ES-ETH (2019)
17. Mazurkiewicz, A.W.: Basic notions of trace theory. In: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. LNCS, vol. 354, pp. 285–363. Springer (1988). https://doi.org/10.1007/BFb0013025
18. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (1999). https://doi.org/10.1007/978-3-662-03811-6
19. Reisig, W.: Petri Nets: An Introduction, Monographs in Theoretical Computer Science. An EATCS Series, vol. 4. Springer (1985). https://doi.org/10.1007/978-3-642-69968-9
20. Saraph, V., Herlihy, M.: An empirical study of speculative concurrency in Ethereum smart contracts. CoRR **abs/1901.01376** (2019), http://arxiv.org/abs/1901.01376
21. Szabo, N.: Formalizing and securing relationships on public networks. First Monday **2**(9) (1997), http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/548
22. Young, J.: CryptoKitties sales hit $12 million, could be Ethereum's killer app after all. https://cointelegraph.com/news/cryptokitties-sales-hit-12-million-could-be-ethereums-killer-app-after-all (2017)