Abstract Ledger Specifications

Polina Vinogradova[1], James Chapman[1], Andre Knispel[1], Orestis Melkonian[1], and Alexey Sorokin[1]

[1] IOG, firstname.lastname@iohk.io
[2] University of Edinburgh, orestis.melkonian@ed.ac.uk

Abstract. Ensuring deterministic behaviour in distributed blockchain ledger design matters to end users because it allows for locally predictable fees and smart contract evaluation outcomes. In this work we begin by establishing a formal definition of an abstract ledger together with a ledger update procedure. We use this model as a basis for formalizing and studying several definitions of the collection of properties colloquially classified as determinism. We investigate the relations between these properties, and the implications each of these has for design and static analysis of ledgers, giving simple but illustrative examples of ledgers with and without these properties. We discuss how these ideas can be applied to realistic, deployable ledger design and architecture.

Keywords: blockchain · formal methods · ledger

1 Introduction

In the context of blockchain transaction processing and smart contract execution, determinism is usually taken to mean something like "the ability to predict locally, before submitting a transaction, the on-chain result of processing that transaction and its scripts". This is an important aspect of ledger design because users care about being able to accurately predict fees they will be charged, rewards they will receive from staking, outcomes of smart contract executions, etc. before submitting transactions. The purpose of this work is to formalize this property of ledgers, and study the constraints under which it can be guaranteed, thereby providing analysis tools and design principles for building ledgers whose transaction processing outcomes can be accurately forecast.

Blockchain ledger and consensus design relies on the property that replaying a its transaction/block history produces the same output every time. We refer to this property as *historic determinism*. The focus of in this work is *predictive determinism*, which focuses on reasoning about the changes a transaction makes, and when these changes are themselves predictable, without needing to know the state to which they will be applied.

The difficulty in predicting the exact on-chain state to which transactions/blocks will be applied is due to unpredictable network propagation of transactions, consensus design, rollbacks, malicious actors, etc. Full predictive determinism, which we will henceforth just call determinism, guarantees if a transaction is valid, the variations in the state due to the reasons mentioned above will not affect how the transaction is processed.

In this work, we first specify a ledger model, then give some definitions of what it means for it to be deterministic. We also consider real-world constraints on ledger design that prevent full determinism from being implementable, and give an analysis of how to treat such less-deterministic ledgers in a useful way.

Previous work on predictive determinism, with a similar ledger construction as we present here, is found in [**?**]. There are notable differences, such as a fixed set of observables, no analysis of transaction failure or analysis of determinism for the cases of dropped transactions, and the focus is more on use of existing ledgers in a deterministic way, rather than ledger design principles. A similar construction to our ledger definition is presented in [**?**]. However, the focus there is on the ordering of state updates to optimize the execution of a particular consensus algorithm. Another related study of commutativity of state transitions is [**?**].

2 The Abstract Ledger Model

We introduce our ledger model and define some basic structure.

Polina: make notes of assumptions of finiteness or at least classical logic?

2.1 What is a ledger?

A ledger specification is given by the types and functions in Figure 1. We use the notation $\odot$ to suggest right monoid action of transaction lists on states. The monoid operation on lists is concatenation, which is compatible with the state update function $\odot$.

Ledger Specification Types

$$\begin{array}{rcll} \mathsf{Tx} & \in & \mathsf{Set} & \text{The type of the transactions} \\ \mathsf{State} & \in & \mathsf{Set} & \text{The type of the ledger state} \end{array}$$

Ledger Specification Functions

$$\begin{array}{rcll} \mathsf{update} & \in & \mathsf{State} \to \mathsf{Tx} \to \mathsf{State} & \text{Updates state with a given transaction} \\ \mathsf{initState} & \in & \mathsf{State} & \text{The initial state of the specification} \end{array}$$

Ledger Specification Notation

$$s \odot tx = \mathsf{update}\ s\ tx$$
Update state s with transaction tx

$$s \odot [tx_1, ..., tx_n] = \mathsf{update}\ (...\ (\mathsf{update}\ (\mathsf{update}\ s\ tx_1)\ tx_2)\ ...)\ tx_n$$
Update state s with transaction tx

Compatibility

$$\mathsf{respectsConcat} : \forall s, txs1, txs2,\ s \odot txs1 \odot txs2 = s \odot (txs1 + + txs2)$$
List concatenation is compatible with $\odot$

Fig. 1: Ledger specification types and functions

2.2 Error states

Given a ledger specification, we can define the following additional structure, see Figure 4. A ledger may have a collection of *error states* Err, each of which having only one possible transition out of it - back to itself. We will sometimes abuse notation by writing $s = \mathsf{Err}$ instead of $s \in \mathsf{Err}$.

The collection of *valid states* V contains all states reachable from the initial state, including any error states, if those are reachable. Note here that we abuse the notation $s \odot tx$, by using it for both $s \in \mathsf{State}$ and $s \in \mathsf{V}$.

Error States and Valid States

$$\mathsf{Err} \in \mathbb{P}\ \mathsf{State}$$
$$\mathsf{Err} = \{\ s \mid \forall\ tx,\ \mathsf{update}\ s\ tx = s\ \}$$
Collection of error states

$$\mathsf{V} \in \mathsf{Set}$$
$$\mathsf{V} = \{\ s \mid \exists\ lstx \in [Tx],\ \mathsf{initState} \odot lstx = s\ \}$$
Set of all valid states

Fig. 2: Ledger specification additional structure

2.3 Threads

We call a *thread* in a ledger the pair (T, π, e) of a target type and a function from the ledger state to T, as specified in Figure 3. Threads will be used to study the evolution of substates of the ledger. We can consider these as "observables" within the ledger state.

A thread (T, π, e) induces a ledger to which we refer to as a T-subledger of the original ledger. To construct this subledger, we specify its transaction type as a pair of a transaction tx and the larger state s to which it is being applied, denoted by tx_s. We define the update function and initial state of the T-subledger in terms of data in the bigger ledger, as in Figure 3.

Threads structure

$\mathsf{T} \in \mathsf{Set}$
The type of a thread

$\pi \in \mathsf{State} \to \mathsf{T}$
Function computing the value of a ledger thread at a given state

$e \in \{ \pi\, s \mid s \in \mathsf{Err} \}$
The error that occurs when the update cannot be applied because of mismatched state

Thread subledger

$\mathsf{initState}_\mathsf{T} \in \mathsf{T}$
$\mathsf{initState}_\mathsf{T} = \pi\ \mathsf{initState}$
Initial state of the subledger defined by the thread T

$\odot_T \in \mathsf{T}(\mathsf{State} \times \mathsf{Tx}) \to \mathsf{T}$
$$q\ \odot_T\ (s, tx) = \begin{cases} e & \text{if } q \neq (\pi\, s) \\ \pi\ (s \odot tx) & \text{otherwise} \end{cases}$$
Update function of a T-ledger

Thread properties

$\mathsf{sameErrs} = \forall\, s \in \mathsf{V}, s \in \mathsf{Err} \Rightarrow \pi\, s \in \mathsf{Err}$
Any subledger specified by a thread enforces mapping errors to errors

Fig. 3: Ledger threads

2.4 Ledger and state comparison relations

To reason about ledger determinism, we would like to be able to specify the difference between and to compare valid states. We begin by re-stating our ledger update function as a tertiary relation rather than a two-argument function. Let us define such a relation $\mathcal{LR}$, see Figure 4.

We would like to describe arbitrary paths between any two valid non-Err states. For this we define the relation $\mathcal{UR}$, which is similar to $\mathcal{LR}$, but each of the transactions in the transition across a sequence valid states can appear either in the forward direction $s_i \odot tx = s_{i+1}$ (specified as $(s_i, [(\mathsf{True}, tx)], s_{i+1})$), or in the backward direction, $s_{i+1} \odot tx = s_i$, specified as $(s_i, [(\mathsf{False}, tx)], s)$.

We use the following notation for representing triples that are in the $\mathcal{UR}$ relation :

$$\langle s, utxs, s' \rangle \quad \Rightarrow \quad (s, txs, s') \in \mathcal{UR}$$

We introduce the notation $\boxdot$ in Figure 4, which allows us to perform an

We define the *delta* of two valid ledger states, denoted $\Delta\ (s, s')$, to be the collection of paths from s to s' in the undirected graph formed by valid, non-Err states as vertices and edges labelled by pairs of a transaction transitioning between the two states, and the direction in which it goes, expressed as a boolean. This collection is never empty, since $[s, ..., \mathsf{initState}, ..., s'] \in \Delta\ (s, s')$. In some sense, the collection of fall paths between them. Note here that distinct transactions (and transaction sequences) may give the same state update, so that $s \odot txs = s' \odot txs$ for some distinct s, s'.

We introduce the relation $* =$ to compare states. Two states satisfy this relation if they are equal, or one of them is an Err state. Note that this relation is not transitive, as if $s \neq s'$, it still holds that $s* = \mathsf{Err}* = s'$.

We introducte the relation $= *$ to compare collections of transition sequences out of two valid states. Two collections are $= *$-equal whenever they both contain the same sequences of transitions, except for possibly those sequences in either that lead to an Err state. Note that this relation is also not transitive, since

Polina: example

3 Deterministic Ledgers

Here we give definitions of determinism and compare them.

3.1 Thread-determinism

We first introduce the most intuinitve concept of determinism in Figure 5. A ledger specification in which the threadDeterminism contraint is satisfied guarantees that the update of any substate of the ledger is independent of other data on the ledger external to the substate. That is, when a substate in a (valid) ledger state s is updated by some transaction, it will be updated to the same substate in any state s' where the original substate is the same as in s.

We say a single thread (T, π) is deterministic whenever $\mathsf{threadDeterminism}\ \mathsf{T}\ \pi$ holds.

Polina: this is the kind of determinism that we use in showing deterministic script validation

Proposition. See the *Thread-determinism proposition for transaction sequences* in Figure 5.

Proof. By induction.

(i) Base case, $txs = []$:

$$\pi\ s \ = \ \pi\ s' \quad \Rightarrow \quad \pi\ s \ = \ \pi\ s'$$

(ii) Inductive step for $txs + +tx$: we assume that,

$$\pi\ s \ = \ \pi\ s',\ \pi\ (s \odot txs) * = \ \pi\ (s' \odot txs)$$

Then, by threadDeterminism at states $\pi\ (s \odot txs)$ and $\pi\ (s' \odot txs)$,
we conclude the result,

$$\pi\ (s \odot (txs + +tx)) \ = \ \pi\ ((s \odot txs) \odot tx) * = \ \pi\ ((s' \odot txs) \odot tx) \ = \ \pi\ (s' \odot (txs + +tx))$$

3.2 Delta-determinism

Delta-determinism is a constraint on a ledger specitication that guarantees that the changes made by a transaction sequence txs to some state s, expressed as the collection of non-Err transactions that all perform the same state update, are the same as the changes made by txs to any other state s'.

This definition of determinism quantifies over all paths in the transition graph, making proving this property directly intractable.

3.3 Order-determinism

The definition in Figure 7 is given in terms of the traces leading to a particular ledger state. Specifically, it is concerned with the independence of the ledger state from the order of the transactions in the trace that has lead to any given state.

3.4 Aspirational results

(i) State and thread det's are equivalent
(ii) Both imply order-det
(iii) Injectivity (questionable, probably at least needs additional preconditions) :
$\forall s, s', txs, s \odot txs = s' \odot txs \Rightarrow s = s'$)

4 Ledger differentials

Theory of changes [**?**] attempt.

Cech cohomology.

5 Categories of Ledgers

There are different categories one may construct out of a ledger specification.

Proposition : Valid states as pullbacks in Set**.** Suppose $\mathcal{P}$ is a pullback in Set,

$$\begin{array}{ccc} \mathcal{P} & \xrightarrow{\pi_2} & [\mathsf{Tx}] \\ {\scriptstyle \pi_1}\downarrow & \lrcorner & \downarrow{\scriptstyle \mathsf{initState}\odot\_} \\ \mathsf{State} & \xrightarrow{\mathsf{id}_{\mathsf{State}}} & \mathsf{State} \end{array}$$

or a given ledger $L = (\mathsf{State}, \mathsf{Tx}, \odot, \mathsf{initState})$. Then the set of all valid states $\mathsf{V} = \pi_2\mathcal{P}$.

Proof. Recall that

$$\mathsf{V} \;=\; \{\; s \;\mid\; \exists\, lstx \;\in\; [Tx],\; \mathsf{initState} \;\odot\; lstx \;=\; s \;\}$$

Recall that a pullback in Set is defined by

$$\mathcal{P} \;=\; \{\; (txs, s) \;\mid\; \mathsf{initState} \;\odot\; txs \;=\; s \;\}$$

Now, for any $s \in \mathsf{V}$, we pick a txs such that $\mathsf{initState} \;\odot\; txs \;=\; s$, since such a txs exists. So, $(txs, s) \in \mathcal{P} \;\Rightarrow\; s \in \pi_2\mathcal{P}$.

For any $s \in \pi_2\mathcal{P}$, there exists $(txs, s) \in \mathcal{P}$ such that $\pi_2(txs, s) = s$, and $\mathsf{initState} \;\odot\; txs \;=\; s$, which proves the result.

5.1 Single-ledger category

Given a ledger $L = (\mathsf{State}, \mathsf{Tx}, \odot, \mathsf{initState})$, we define the category SL, constructed from a single ledger specification's types, transitions, and initial state, in the following way :

(i) Objects : $\mathsf{Obj}\ \mathsf{SL} = \mathsf{State}$

(ii) Morphisms : $\mathsf{Hom}\ S1\ S2 = \{\ \_ \odot txs \mid txs \in [\mathsf{Tx}]\ ,\ S1 \odot txs = S2\}$

(iii) Identity : $\mathsf{id}_{S1} = [] \in \mathsf{Hom}\ S1\ S1$

(iv) Composition : $\forall\ txs1,\ txs2, S$
$(\_ \odot txs2) \circ (\_ \odot txs1)(S) = (\_ \odot txs2)(S \odot txs1) = (S \odot txs1) \odot txs2$
$= S \odot (txs1 + + txs2) \Rightarrow (\_ \odot (txs1 + + txs2)) \in \mathsf{Hom}\ S1\ S3$

(v) Associativity : holds because the monoid action respects list concatenation in the list monoid (ie. respectsConcat)

5.2 Category of all ledgers

We first introduce some notions needed to describe the category of all ledgers $\mathcal{L}$ and maps between them that preserve ledger structure.

Let Set_* denote the category of pointed sets with maps $f \in \mathsf{Hom}(\{*\} \to S,\ \{*\} \to Q)$, such that $(* \mapsto x \in S, s \in S) \mapsto (* \mapsto f(x), f(s)) \in Q$ for some set map $f : S \to Q$.

Let Monoid denote a category of monoids $(M : \mathsf{Set},\ \mu : \mathsf{Set} \times \mathsf{Set} \to \mathsf{Set},\ e : M)$ and monoid action- and identity-preserving homomorphisms. We use (M, μ, e) and M interchangeably here.

Let $\langle \_ \rangle : \mathsf{Monoid} \dashv \mathsf{Set} : \mathcal{U}$ be the free-forgetful adjunction for monoids.

Let us now define a functor Γ :

$$\Gamma : \begin{cases} \mathsf{Set}_* \times \mathsf{Monoid} \to \mathsf{Set}_* \times \mathsf{Monoid} \\ (i : \{*\} \to S, M) \mapsto (\{*\} \to S \times \mathcal{U}M, M) \\ (f \times g : (i : \{*\} \to S, M) \to (\{*\} \to Q, N)) \mapsto ((\langle f, \mathcal{U}\ g \rangle, g)(\{*\} \to (S \times \mathcal{U}M), M) \to (\{*\} \to (Q \times \mathcal{U}N), N)) \end{cases}$$

We now define the natural transformations $\gamma : \Gamma^2 \to \Gamma$ and $\eta : \mathbb{1}_{\mathsf{Set} \times \mathsf{Monoid}} \to \Gamma$.

Note that

$$\Gamma(i : \{*\} \to S, M) = (\langle i, \mathcal{U}e \rangle : \{*\} \to S \times \mathcal{U}M, M)$$

and

$$\Gamma^2(i : \{*\} \to S, M) = (\langle i, \mathcal{U}e, \mathcal{U}e \rangle : \{*\} \to S \times \mathcal{U}M \times \mathcal{U}M, M)$$

So that we define

$$\gamma_{(i:\{*\} \to S, M)} : \begin{cases} \Gamma^2(i : \{*\} \to S, M) \to \Gamma(i : \{*\} \to S, M) \\ (* \mapsto (i, \mathcal{U}e, \mathcal{U}e), (s, m_1, m_2), m) \mapsto (* \mapsto (i, \mathcal{U}(\mu e e)), (s, \mathcal{U}(\mu m_1 m_2), m) \end{cases}$$

$$\eta_{(i:\{*\} \to S, M)} : \begin{cases} (i : \{*\} \to S, M) \to (\langle i, \mathcal{U}e \rangle : \{*\} \to S \times \mathcal{U}M, M) \\ (* \mapsto i, s, m) \mapsto (* \mapsto (i, \mathcal{U}e), (s, \mathcal{U}e)m) \end{cases}$$

The data (Γ, γ, η) is a monad on $\mathsf{Set}_* \times \mathsf{Monoid}$. A Γ-algebra is made up of $(i : \{*\} \to S, M)$ together with the following operation, induced by a monoid action $\odot : S \times M \to S$

$$\odot_{(i:\{*\} \to S, M)} : \begin{cases} \Gamma(i : \{*\} \to S, M) \to (i : \{*\} \to S, M) \\ (* \mapsto (i, \mathcal{U}e), (s, m_1), m) \mapsto (* \mapsto i, (s \odot m_1), m) \end{cases}$$

We define the category of all ledgers, $\mathcal{L}$, in terms of these structures. Recall here that the morphisms in the product category with objects $\mathsf{Set}_* \times \mathsf{Monoid}$ are those that preserve $*$ in the first coordinate, and monoid structure in the second.

(i) Objects :
$\mathsf{Obj}\ \mathcal{L} = \mathsf{Set}_* \times \mathsf{Monoid}$

(ii) Morphisms are a subset of the morphisms of the product morphisms between objects of $\mathsf{Set}_* \times \mathsf{Monoid}$ which contains only those maps that preserve each $\odot_{(i:\{*\}\to S,M)}$:

$$\mathsf{Hom}\ (i : \{*\} \to S, M)\ (i : \{*\} \to Q, N) = \{f\ :\ (i : \{*\} \to S, M) \to (i : \{*\} \to Q, N)\ \mid \odot_{(i:\{*\}\to Q,N)}\ \circ\ (\Gamma\, f)\ =\ f \circ \odot_{(i:\{*\}\to S,M)}\}$$

(iii) Identity : $\odot_{(i:\{*\}\to Q,N)}\ \circ\ (\Gamma\, \mathbb{1}_{(i:\{*\}\to S,M)})$
$= \odot_{(i:\{*\}\to Q,N)}\ \circ\ \mathbb{1}_{\Gamma\ (i:\{*\}\to S,M)}$
$= \odot_{(i:\{*\}\to Q,N)}$
$= \mathbb{1}_{(i:\{*\}\to S,M)} \circ \odot_{(i:\{*\}\to Q,N)}$

(iv) Composition (we drop the subscript of $\odot$ from here onwards) :
$\forall\ f \in \mathsf{Hom}\ (i : \{*\} \to S, M)\ (i : \{*\} \to Q, N),\ g \in \mathsf{Hom}\ (i : \{*\} \to Q, N)\ (i : \{*\} \to P, R),$
$(g \circ f)\ \circ\ \odot\ =\ g(f \circ \odot) =\ g\ (\odot \circ (\Gamma f))$
$=\ g\ (\odot \circ (\Gamma f))$
$=\ \odot \circ (\Gamma g) \circ (\Gamma f)$
$=\ \odot \circ (\Gamma(g \circ f))$

(v) Associativity : inherited from maps in $\mathsf{Set}_* \times \mathsf{Monoid}$.

$\boxdot \in \mathsf{State} \to (\mathsf{Bool},\ \mathsf{Tx}) \to \mathbb{P}\ \mathsf{State}$
$s \boxdot (b,\ tx) = \{\ s' \mid b \Rightarrow s \odot tx = s' \wedge \neg b \Rightarrow s' \odot tx = s\}$
Notation of applying or unapplying a transaction

$\mathcal{LR} \in \mathbb{P}\ (\mathsf{State} \times [\mathsf{Tx}] \times \mathsf{State})$
$\mathcal{LR} = \{\ (s \in \mathsf{V}, txs \in [\mathsf{Tx}], s' \in \mathsf{V}) \mid \mathsf{Err} \neq s \odot txs = s' \neq \mathsf{Err}\}$
All non-error transitions between valid ledger states

$\mathcal{UR} \in \mathbb{P}\ (\mathsf{State} \times [(\mathsf{Bool}, \mathsf{Tx})] \times \mathsf{State})$
$\mathcal{UR} = \{\ (s, [], s')\ \mid \mathsf{Err} \neq s = s'\} \cup \{\ (s \in \mathsf{V}, [(b_1, tx_1), ..., (b_k, tx_k)] \in [\mathsf{Tx}], s' \in \mathsf{V})\ \mid$
$\exists\ [s_1, ..., s_{k+1}],\ s = s_1 \neq \mathsf{Err},\ s' = s_{k+1} \neq \mathsf{Err},\ k \geq 1, \forall 1 \leq i \leq k,$
$((b_i \wedge (s_i, [tx_i], s_{i+1}) \in \mathcal{LR}) \vee (\neg b_i \wedge (s_{i+1}, [tx_i], s_i)\ \in\ \mathcal{LR})\ \}$
All non-error transitions in either direction between valid ledger states

$\boxdot \in \mathsf{State} \to [\mathsf{Bool},\ \mathsf{Tx}] \to \mathbb{P}\ \mathsf{State}$
$s \boxdot [(b_1,\ tx_1), ..., (b_k,\ tx_k)] = (s \boxdot)$
Notation of applying or unapplying a transaction

$\Delta \in (\mathsf{V} \times \mathsf{V}) \to \mathbb{P}\ [(\mathsf{Bool}, \mathsf{Tx})]$
$\Delta(s, s') = \{\ utxs \mid (s, utxs, s') \in \mathcal{UR}\ \}$
Collection of paths from s to s' in two-directional transaction graph

$(* =) \in \mathsf{State} \to \mathsf{State} \to \mathsf{Bool}$
$s * =\ s' = (s = s') \vee (s \in \mathsf{Err}) \vee (s' \in \mathsf{Err})$
Relation containing pairs of identical states or a state and an error state

$(= *) \in \mathsf{V} \to \mathsf{V} \to \mathbb{P}\ [(\mathsf{Bool}, \mathsf{Tx})] \to \mathbb{P}\ [(\mathsf{Bool}, \mathsf{Tx})] \to \mathsf{Bool}$
$stx1 =*_{s1,s2} stx2 = (\forall\ lstx \in stx1,\ s1 \boxdot lstx \subseteq \mathsf{Err} \vee lstx \in stx2)$
$\wedge\ (\forall\ lstx \in stx2,\ s2 \boxdot lstx \subseteq \mathsf{Err} \vee lstx \in stx1)$
Relation compating sets of transitions from two different states

Fig. 4: Comparisons of states and state transition collections

Thread-determinism definition on a ledger specification

$$\begin{aligned}\mathsf{threadDeterminism} \;=\; & \forall\, \mathsf{T},\ \pi \in \mathsf{State} \to \mathsf{T},\\ & \forall\, (s,\ s' \in \mathsf{V}),\ t \in \mathsf{T},\ tx \in \mathsf{Tx},\ \pi\, s = \pi\, s',\\ & \pi\, (s \odot tx) \ast = \pi\, (s' \odot tx)\end{aligned}$$

Thread-determinism proposition for transaction sequences

$$\begin{aligned}\mathsf{threadDeterminism} \;=\; & \forall\, \mathsf{T},\ \pi \in \mathsf{State} \to \mathsf{T},\\ & \forall\, (s,\ s' \in \mathsf{V}),\ t \in \mathsf{T},\ txs \in [\mathsf{Tx}],\ \pi\, s = \pi\, s',\\ & \pi\, (s \odot txs) \ast = \pi\, (s' \odot txs)\end{aligned}$$

Fig. 5: Thread-determinism

State-determinism constraint on a ledger specification

$$\begin{aligned}\mathsf{deltaDeterminism} \;=\; & \forall (s\ s' \in \mathsf{V})\ (txs \in [\mathsf{Tx}]),\\ & \Delta(s,\ s \odot txs) \ast =_{s,s'} \Delta(s',\ s' \odot txs)\end{aligned}$$

Fig. 6: State-determinism

Order-determinism constraint on a ledger specification

$$\begin{aligned}\mathsf{orderDeterminism} \;=\; & \forall\, (s \in \mathsf{V})\ (txs \in [\mathsf{Tx}]),\\ & txs' \in \mathsf{Permutation}\ txs,\\ & s \odot txs \neq \mathsf{Err} \neq s \odot txs'\\ & \Rightarrow\ s \odot txs = s \odot txs'\end{aligned}$$

Fig. 7: Order-determinism