Optional Monitoring for Long-Lived Transactions

Joshua Ellul and Gordon J. Pace

Centre for DLTs
Department of Computer Science,
University of Malta, Malta

VORTEX 2021

Smart Contracts

- Operationalise the regulation of behaviour between different parties with no centralised point of trust.
- Residing on a blockchain, they have to be passive, in that they perform some action on being invoked, after which they release control to the rest of the world.
- Smart contracts are really about interaction.

Smart Contracts and Interaction

- One shot agreements do not need decentralisation — *“John gives Jane one euro”* is just carried out.
- Temporal agreements may need such decentralisation. Consider *“John deposits a token in escrow which is only given to Jane once she pays one euro.”*
  - Jane wants to be sure she will get the token after paying.
  - John to be sure that his token will not be given to Jane before she pays.
- Smart contracts are really useful in the latter case, where a unit of computation lives across different interactions (invocations).
- Most smart contracts really encode *long-lived transactions*.

Long-Lived Transactions

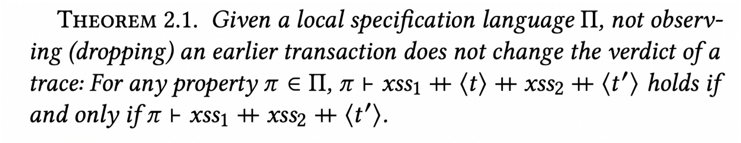
- Transactions which may last for too long a period to allow for locking of resources, but which could lead to an inconsistent internal state if the resources are released too early.
- Frequently used in financial transaction systems, when some parts of a transaction need manual human input such as checking identity of a client or awaiting a bank to approve a transfer.
- In practice, one has a hierarchy of transactions being made up of finer-grained transactions which in turn are made up of. . . made up of atomic transactions.
- Typically one also includes the concept of compensations to make up for successful parts of the transaction if failure happens later on half way through a long-lived transaction.

Verification of Transaction Systems

- Specification of such systems would typically include a comparatively small number of global soundness properties.
- Most of the specification is typically made up of parameterised properties universally quantified over some level of a long lived transaction:
    - *"Users may not perform a transfer before being approved by an administrator"* $\equiv$
      $\forall u : \mathit{User} \cdot$ *u.transfer cannot be called before u.approve*
    - *"Users may not transfer more than 500 euros in a single login session"* $\equiv$
      $\forall s : \mathit{Session}\ \cdot \sum\{v \mid s.\mathit{transfer}(v)\} \leq 500$
- One of the good things about these properties is that they are semantically independent of each other.

Verification of Independent Properties

- Events from different properties (LLTs) commute meaning that if we can gather the events of a LLT in a single contiguous subtrace.
- Not monitoring/verifying an LLT will not change the judgement about other LLTs.
- We can thus offer a choice as to whether or not a property should be runtime verified for a particular LLT.



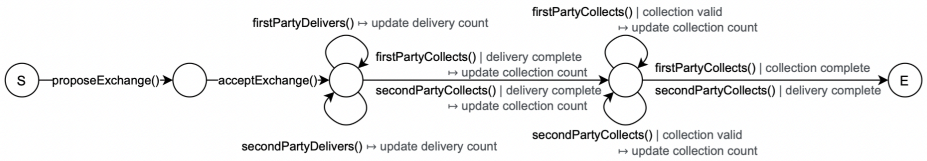
THEOREM 2.1. *Given a local specification language* Π*, not observing (dropping) an earlier transaction does not change the verdict of a trace: For any property* $\pi \in \Pi$*,* $\pi \vdash xss_1 \mathbin{+\!\!+} \langle t \rangle \mathbin{+\!\!+} xss_2 \mathbin{+\!\!+} \langle t' \rangle$ *holds if and only if* $\pi \vdash xss_1 \mathbin{+\!\!+} xss_2 \mathbin{+\!\!+} \langle t' \rangle$.

Implementation and Evaluation

- Optional monitoring built for Solidity smart contracts with ContractLarva.
- As a use case we use a token exchange smart contract in which:
  - Any party may propose an exchange (*n* tokens of one type for *m* tokens of another with a chosen second party)
  - If accepted by the counter-party, starts the exchange, in which...
  - The smart contract accepts the tokens from both parties and then...
  - Allows them to be collected by the parties (thus swapping ownership).

Implementation and Evaluation

- The expected behaviour displayed below is taken to be the property to monitor


S
proposeExchange()
acceptExchange()
firstPartyDelivers() ↦ update delivery count
secondPartyDelivers() ↦ update delivery count
firstPartyCollects() | delivery complete ↦ update collection count
secondPartyCollects() | delivery complete ↦ update collection count
firstPartyCollects() | collection valid ↦ update collection count
secondPartyCollects() | collection valid ↦ update collection count
firstPartyCollects() | collection complete
secondPartyCollects() | delivery complete
E

- The initiator gets to choose whether or not it is a monitored property, reducing monitoring overheads of a full LLT execution from 7.8% to 2.8%.

Is it a Valid Evaluation?

- Complexity-wise it is pretty straightforward — switching monitoring off results in constant overheads per call in the long lived transaction (a boolean check to see whether monitoring is on or off) as opposed to the complexity of the monitoring code.
- Overheads would change depending on various variables:
  - How expensive monitoring the property is (the more expensive it is, the more we gain switching it off)
  - How expensive the underlying system is to compute (the more expensive it is, the lower the percentage overheads)
  - How many steps there are in a long lived transaction (the more there are, the more times we have to pay for the check whether monitoring is on or off).
- So what is a fair evaluation? (And this is a question for runtime verification tools in general.)

Conclusions and Future Work

- Monitoring is not always required for slices of the system behaviour. Parties may know and trust each other, the potential losses may be low, etc.
- A clean way of slicing the behaviour is that of long lived transactions.
- Future directions:
  - Reduce overheads further by having an underlying monitoring-aware virtual machine.
  - Give a blame and economic gain/loss semantics to enable justification of monitoring choice.
  - Use compensations to enable undoing prior computation if a transaction fails half-way through.
  - Language-level transaction handling