Optional Monitoring for Long-Lived Transactions

Joshua Ellul*
Gordon J. Pace*
joshua.ellul@um.edu.mt
gordon.pace@um.edu.mt
Centre for Distributed Ledger Technologies, University of Malta
Msida, Malta

Abstract

Runtime monitoring comes at a runtime cost. Overheads induced by monitoring and verification code may be necessary, and yet prohibitive in certain circumstances. When verification is local to a single unit of execution in a system, one can choose whether or not to monitor based on the risk of that individual unit. In this paper, we propose a monitoring and verification approach for a class of long-lived transaction-based systems whose execution can be partitioned into separate subtraces, one for each such transaction, and which are independent of each other from a correctness perspective. We focus on the use of this approach for the monitoring of smart contracts on distributed ledger technologies to show how we can reduce overheads in this manner.

CCS Concepts: • **Software and its engineering** → *Formal language definitions*; *Specification languages*; *Formal software verification*; *Software defect analysis.*

Keywords: formal methods, runtime verification, long-lived transactions, smart contracts

ACM Reference Format:
Joshua Ellul and Gordon J. Pace. 2021. Optional Monitoring for Long-Lived Transactions. In *Proceedings of the 5th ACM International Workshop on Verification and mOnitoring at Runtime EXecution (VORTEX '21), June 12, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3464974.3468450

1 Introduction

Blockchain and other distributed ledger technologies (DLTs) have enabled the possibility of having trusted code execution without the need for trusted parties. Smart contracts are nothing but computer programs in the most traditional sense of the word but differ, and take their name from the computational model they are executed on[1]. By ensuring that the code is faithfully executed in an untampered manner without a centralised party having control over its execution is the key distinguishing element, making them ideal to regulate behaviour between parties.

Being code, however, immediately raises the question of correctness. Even more so when such code is meant to regulate parties' behaviour. It is useless to have untampered computation of incorrect code. Even worse is that a key aspect required for trusted execution is that the code cannot be indiscriminately changed, and the immutability of the code is an important feature of smart contracts. Immutable code means that the effects of bugs are even more severe, and although one can include code in a smart contract to change its logic for instance by consensus of the parties, this is not always a solution, since the party benefiting from a bug may refuse to agree to a change.

Due to the importance of smart contract correctness, there has been much work on their verification. Many approaches use static analysis to ensure correctness a priori, an approach justified by the need to deploy only correct contracts but also, from a pragmatic point-of-view, by the fact that many smart contracts are relatively small pieces of code [1, 8, 13].

However, static analysis does not always scale up to deal with smart contract business logic, and runtime verification solutions have also been built to allow for the monitoring of related properties that cannot be proven statically [2, 12]. Given the immutable nature of smart contracts, synchronous and online monitoring has to be planned and deployed before the smart contract is instantiated on the blockchain[2]. However, this is not much different to monitoring of traditional systems. The main Achilles heel of runtime verification remains that of overheads induced by computation dedicated to monitoring, and with smart contracts running on public blockchains this becomes an even more serious concern

*Both authors contributed equally to this research.

[1]Some add that smart contracts are also different in that they also interact directly with the underlying blockchain transactions, handling the transfer of digital assets. However, this is more a side effect of the execution model using such platforms than part of their very nature.
[2]One can build in generic monitoring code into the immutable contract, allowing for updateable monitors e.g. as discussed in [2], but the monitoring infrastructure code would still have to be built in prior to instantiation on the blockchain.

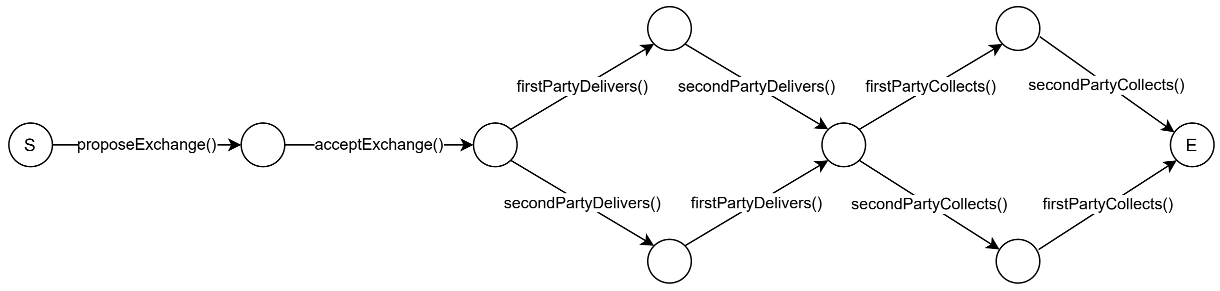

Figure 1. Long-lived transaction of token exchange

since, typically, smart contract users are required to pay (cryptocurrency) for the execution of invoked computational logic — by doing so public DLTs avoid attacks and bugs that would result in extensive computation which would keep nodes busy and unable to attend to other pending transactions [14]. Accompanied by an increase in these execution costs makes runtime verification and other dynamic analysis techniques less attractive, and means of lowering such overheads have increased importance.

At this stage, it is worth noting how smart contract runtime behaviour takes place and is typically structured. Smart contracts are essentially made up of executable code, with functions which can be invoked. In order to invoke such functions from outside the blockchain, one initiates a type of transaction on the blockchain instructing that function to be executed by the miners. Although individual invocations of smart contract functions happens through blockchain transactions, many of them have the notion of logical transactions whose lifetime involve the invocation of multiple blockchain transactions. Such logic spanning multiple atomic transactions (in our case, the invocations of functions through blockchain transactions) is referred to as a *long-lived transaction* [10]. The following example should help clarify terminology.

Example 1. *For instance, a smart contract may allow parties to initiate a token exchange and carry it out over multiple function calls to the smart contract performed by the parties involved. The lifetime of such an exchange can be visualised in Figure 1. The transaction starts (in the state marked by S) by one of the parties proposing an exchange, possibly identifying the address of the party to trade with, and the amounts and type of tokens to be exchanged between the two parties, after which the second party may accept. The parties would then be expected to submit to the smart contract the tokens they promised in either order, only after which may the parties take back their share before the full transaction ends (in the state marked E). It is worth noting that using this notation we will assume that from any state functions not appearing on outgoing transitions should be refused. Also note that one can have looping behaviour (e.g. if we want to allow the parties to deposit or withdraw their tokens in part). Finally note that we have kept the property simple and we do not cover certain cases e.g. when one party deposits their share but the second party fails to do so, in which case a timeout may have to be imposed after which the first party may pull out of the deal.*

It is worth noting that in the lifetime of a smart contract regulating such exchanges one can have multiple such exchanges taking place, including ones being carried out concurrently. This notion of transactions spanning over multiple system execution units (e.g. function calls) has long been used in systems such as financial transaction systems, and are typically called *long-lived transactions* [10], as we will refer to them in the rest of this paper.

One commonly found aspect of many such transaction-based systems is that the execution and correctness of each individual long-lived transaction is local i.e. independent of that of others ongoing transactions, thus allowing for verification at a per long-lived transaction approach. Note that this is not always the case — for instance, if the smart contract enforces a limit on the number of certain tokens to transact per day for each party, two long-lived transactions involving the same party would not be independent.

However, for independent long-lived transactions one can choose to runtime verify or not upon the initiation of each transaction without impacting the verification of others. On traditional monolithic systems, this can be done, for instance, by sampling transactions or using a risk assessment approach to decide which to monitor, but in the context of smart contracts this is a particularly attractive option. Different long-lived transactions may involve different parties, and it can be left up to them to decide whether they would like their interaction to be runtime verified or whether they feel that the additional cost is not worth paying for that particular extended exchange.

In this short paper we present initial steps towards building such a selective monitoring framework. We present a simplified model of independent long-lived transactions and present how this can be implemented and optimised for smart contracts written in Solidity and deployed on Ethereum [14], with monitoring instrumented using the runtime verification tool ContractLarva [2]. We see this as a first step towards a richer model of dependent transactions enabling more flexible transaction monitoring, and integrating the approach with earlier work we have done on optimisation of selective monitoring at a virtual machine level [6].

2 Localised Monitoring

Switching on and off monitoring can clearly result in monitoring to break for certain properties. For instance, consider a property which states that a transfer of an asset may only happen if a payment approval took place. If the two actions are performed as required but the monitoring is switched off when the payment approval takes place, the monitor will highlight a violation. On the other hand, if the property states that there lies an obligation to perform the transfer if payment is approved and monitoring is switched off when the approval takes place, the system failing to transfer the asset would not be identified as a violation. Stateful specification languages may result in false positives (violations flagged when there was no violation) and false negatives (no violation flagged when one occurred). In order to address this, we make an assumption that the monitoring mode can only be changed between transactions and identify the class of (trace-based) specification languages which ensure that runtime violation identification is both sound and complete for transactions during which monitoring is turned on. In order to formalise these notions, we will make use of the following notation.

Notation 1. *We will write $seq(X)$ to denote finite sequences over items of type X, writing $\langle x_1, x_2 \ldots x_n \rangle$ to write a particular trace. We will write $x : xs$ to denote the list produced when prepending x to xs, and $xs_1 +\!\!+\, xs_2$ to denote the concatenation of two lists. We write $head(xs)$ and $tail(xs)$ to denote the head and the tail of the list, and dually $last(xs)$ and $init(xs)$ starting from the end.*

Definition 1. *Given two lists of lists $xss_1, xss_2 \in seq(seq(X))$, we say that the former is a prefix of the latter: $xss_1 \preceq xss_2$ if either (i) we can add more sequences to xss_1 to obtain xss_2: $\exists xss'_1 \cdot xss_1 +\!\!+\, xss'_1 = xss_2$; or (ii) we can extend the last sequence in xss_1 and also add more sequences to obtain xss_2: $\exists xs, xss'_1 \cdot init(xss_1) +\!\!+\, \langle last(xss_1) : xs \rangle +\!\!+\, xss'_1 = xss_2$.*

We identify sequential transaction-based systems whose trace behaviour corresponds to a number of sequential transactions i.e. transactions cannot occur concurrently.

Definition 2. *Given a sequential transaction system S over observable event alphabet Σ, its runtime trace behaviour, written $[\![S]\!]$, is a sequence of transactions, where each transaction is a sequence of events from Σ: $[\![S]\!] \subseteq seq(seq(\Sigma))$. We will assume that the set of traces of S is closed under prefixes i.e. for any system S, if $xss \in [\![S]\!]$ then $\forall xss' \cdot xss' \preceq xss \implies xss' \in [\![S]\!]$.*

The alphabet can correspond to what is of interest to observe. It may, for instance, simply be function names to indicate the invocation of functions, or annotated function names to allow the observation of entry and exit from a function, or even variable assignments, denoting the value of each variable in that snapshot.

Definition 3. *We will characterise specification language Π with a trace satisfaction relation $\vdash \subseteq \Pi \times seq(seq(\Sigma))$. We will use the term* property *to refer to instances of such a specification language $\pi \in \Pi$.*

Note that we do not limit ourselves to specification languages for which the extension of a failing trace always fails. For instance, consider the case of Π_{inv} corresponding to invariants on variable values. If the alphabet corresponds to variable assignments, we can characterise the property $\pi_{x \leq y}$ saying that the value of variable x never exceeds the value of variable y, by defining $\pi_{x \leq y} \vdash xss$ to hold if $\sigma = \text{last}(\text{last}(xss))$ is well-defined, and $\sigma(x) \leq \sigma(y)$. Note that, with such a definition we would observe every time the invariant is violated.

Definition 4. *A specification language Π with a trace satisfaction relation $\vdash \subseteq \Pi \times seq(seq(\Sigma))$, is said to be localised if for any property $\pi \in \Pi$: $\pi \vdash ts +\!\!+\, \langle t \rangle$ holds if and only if $\pi \vdash \langle t \rangle$.*

It is easy to prove that the specification language of invariants Π_{inv} as defined above is localised. However, it is worth noting that richer logics which handle state within a transaction may also be shown to be localised. For instance, in specification logics which allow for universal quantification over a property such as Quantified Event Automata as used in MarQ [11], or DATEs as used in Larva [3] one may have properties which are to hold for all transactions (i.e. partition the events per transaction), and as long as these transactions do not share any state, it would be possible to prove that they are localised. Similarly, if we know that each transaction will start with an event in which *startTransaction* holds and ends which one in which *endTransaction* holds, the subset of LTL properties of the following form:[3]

$$\Box(startTransaction \Rightarrow \psi \; W \, endTransaction)$$

i.e. *from the moment a long-lived transaction starts, property ψ must hold until the end of the long-lived transaction, or forever if it does not terminate*, can also be shown to be localised.

Local specification languages allow us to drop whole transactions without changing the verdict according to a property.

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 $\pi \in \Pi$, $\pi \vdash xss_1 +\!\!+\, \langle t \rangle +\!\!+\, xss_2 +\!\!+\, \langle t' \rangle$ holds if and only if $\pi \vdash xss_1 +\!\!+\, xss_2 +\!\!+\, \langle t' \rangle$.*

[3]Using standard notation $\Box$ indicating the *always* modality and W indicating *weak until*.

3 Applying to Long-Lived Transactions on Smart Contracts

In order to illustrate the use of the approach we describe in this paper, we will be using a token exchange smart contract as described in Example 1. A smart contract has been built which allows for any user to propose a token exchange by specifying: (i) the counter-party; (ii) the type of and number of tokens s/he is offering; and (iii) the type and number of tokens s/he is expecting from the counter-party. Once the counter-party accepts, both parties may then submit their tokens, only after which the parties may withdraw their share of the exchange. Once a long-lived transaction terminates, another can be triggered, by the same or different parties. Needless to say, the possibility of bugs in the implementation justifies that parties may wish to runtime verify such a long lived transaction.

To illustrate the use of per long-lived transaction monitoring, we will use the automaton shown in Figure 1 as our property — ensuring that any interaction allowed by the implementation follows the presupposed protocol e.g. that neither party can collect before both parties have delivered. The semantics given to such an automaton (as a property) is that (i) the property starts in the initial state when a long-lived transaction starts; (ii) when in any state a function appearing on an outgoing transition is successfully[4] called, it will change the state of the property; and (iii) when either a new long-lived transaction is triggered when another is still active, or when in a state a function other than one appearing on an outgoing transition is successfully called there is a violation of the property. Branching transitions from a state simply indicates that functions decorating either outgoing transition are accepted and allow progress in the automaton e.g. after accepting an exchange, both parties are allowed to deliver first. For the sake of this paper we do not go into how to resolve such violations. With these semantics, since each long-lived transaction initiates a new monitor with no memory of previous transactions, one can show that the logic is local.

If we want to monitor every long-lived transaction, we can see the specification as the property universally quantified over each such transaction. Each function invocation received triggers any active transitions. Using ContractLarva [7], it is straightforward to build such a fully runtime verified version of the smart contract, with the property given in the automaton being used directly as input to ContractLarva[5].

However, we recognise that not every proposed exchange of tokens is critical. Parties may know and trust each other, or the value of the exchanged tokens may be so low that paying additional overhead for verifying that particular long-lived transaction may not be desirable. To handle such a choice, we modify our runtime monitored artifact so that the function(s) triggering a new instance of the property (`proposeExchange()` in this case) are duplicated to allow the party invoking them choosing between a monitored or an unmonitored instance[6]. All monitoring specified in ContractLarva is simply guarded by a condition to trigger if and only if monitoring is switched on for that long-lived transaction. The check just involves verifying whether a monitored long-lived transaction is currently active, making the overhead for unmonitored instances very low.

In order to illustrate how the approach would work on a more complex example, we modified the smart contract to allow the parties to deposit and take out their tokens in batches i.e. not necessarily in one go. The property used for this case is similar to the previous one and is shown in Figure 2. Note that transitions are now annotated as $f \mid c \mapsto a$, which are triggered when function f is successfully executed and condition c is satisfied. Action a will be additional code to execute to support monitoring, for instance in this case we use it to keep track of the total deposited and withdrawn by each of the parties. This property can be used for a more fine-grained token exchange contract, in which the parties need not deposit (and collect) the tokens in one go. Rather, they can do so in smaller batches. For example, consider an agreed upon exchange between party A who will provide 9 tokens of type T, and party B who will provide 5 tokens of type U. The two parties may provide their tokens in smaller chunks, e.g, A delivering 5 of his tokens, then B delivers 3, A delivers 2 then another 1, B delivers 2 (completing his delivery), and finally A delivers 1 token (completing her delivery). Similarly, the collection of tokens can be similarly broken up into steps e.g. B collects 6 tokens, A collects 5, and finally B collects 3. The monitoring condition and action keep count of who has still to deliver (or collect) how many tokens.

Evaluating the gains when one switches off runtime monitoring is not as straightforward as it may seem. It is dependent on multiple things — how expensive the monitoring of the property is (the more expensive, the more we gain when comparing monitored transactions vs. transactions with monitoring switched off), how expensive the logic of underlying system to carry out the long-lived transaction (the more expensive it is, the lower the percentage cost of switched off monitoring which is essentially of constant cost per invocation) and the number of transactions in the long-lived transaction (the more there are, the more times we have to pay the constant boolean check whether monitoring is required). In order to provide a feel to the cost, we have

[4]Function calls may revert, cancelling their effect altogether, which is considered unsuccessful.

[5]The current version of ContractLarva does not natively support replication of monitors, but this was achieved through a straightforward change in the monitoring code produced by ContractLarva.

[6]One can have more sophisticated means of deciding whether or not a long-lived transaction is to be monitored e.g. by having all participating parties take a vote to decide. We plan to investigate such policies in future work.

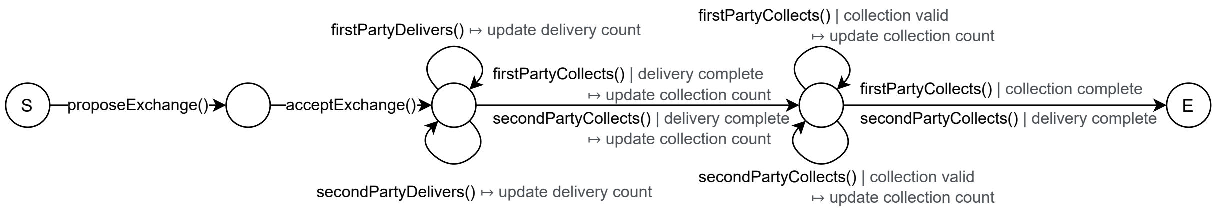
S
proposeExchange()
acceptExchange()
firstPartyDelivers() ↦ update delivery count
secondPartyDelivers() ↦ update delivery count
firstPartyCollects() | delivery complete ↦ update collection count
secondPartyCollects() | delivery complete ↦ update collection count
firstPartyCollects() | collection valid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

Figure 2. Long-lived transaction of token exchange in parts

implemented a token management smart contract and instrumented the property shown in Figure 2. When executing the exchange in chunks as described in the previous paragraph, we observed that while monitoring would have increased the cost of the full long-lived transaction by 7.8%, running the transaction with monitoring switched off would reduce the overheads to 2.8%. This is just over a third of the full monitoring costs, and would not increase if the property were to be more complex i.e. gains would be even higher.

4 Conclusions

In this short paper we have started looking at optional monitoring of independent long-lived transactions. The work presented here just scratches the surface of the opportunities arising from such an approach. We are currently extending this work in a multitude of ways.

Firstly, we are looking at enriching the domain of applicability of the approach. By allowing for interleaved long-lived transactions (i.e. multiple long-lived transactions can be triggered at the same time), we would be able to handle a much richer set of smart contracts. Also, we would like to develop richer dependency models to be able to reason what parts of the monitoring can be turned off without affecting others. Such a dependency relation can be at the level of a full long-lived transaction, but can also be a more fine-grained partial order approach at the monitoring state level, thus still allowing for reduction of overheads at a finer level of granularity. For instance, in the example we gave in the introduction which limited the number of a certain type of token per user per day, one may still partially turn off monitoring of certain long-lived transaction-level properties keeping the monitoring machinery tracking token exchange counts active.

We are also looking at other more effective means to allow for switching on and off monitoring. As we have seen in this paper, one solution is to context switch at the start of each function call depending on whether monitoring is on or off for the ongoing long-lived transaction. However, in previous work [6], we have explored allowing for optional monitoring at the underlying virtual machine level. We intend to look into using such an approach for optional monitoring of long-lived transactions.

An aspect of blockchain-based long-lived transactions which we did not handle in this paper is that of undoing the long-lived transaction as a whole in case of failure midway through its lifetime — typically called *compensations* [9, 10]. In previous work, we have looked at how runtime verification can integrate with such compensatory mechanisms in the context of monolithic systems [4, 5], and we plan to integrate those techniques in the approach we have hereby presented.

References

[1] Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. 2018. Smart Contracts: A Killer Application for Deductive Source Code Verification. In *Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*, Peter Müller and Ina Schaefer (Eds.). Springer, 1–18. https://doi.org/10.1007/978-3-319-98047-8_1

[2] Shaun Azzopardi, Joshua Ellul, and Gordon J. Pace. 2018. Monitoring Smart Contracts: ContractLarva and Open Challenges Beyond. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11237)*, Christian Colombo and Martin Leucker (Eds.). Springer, 113–137. https://doi.org/10.1007/978-3-030-03769-7_8

[3] Christian Colombo and Gordon J. Pace. 2017. Runtime Verification using LARVA. In *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA (Kalpa Publications in Computing, Vol. 3)*, Giles Reger and Klaus Havelund (Eds.). EasyChair, 55–63. http://www.easychair.org/publications/paper/Jwmr

[4] Christian Colombo, Gordon J. Pace, and Patrick Abela. 2010. Compensation-Aware Runtime Monitoring. In *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6418)*, Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Springer, 214–228. https://doi.org/10.1007/978-3-642-16612-9_17

[5] Christian Colombo, Gordon J. Pace, and Patrick Abela. 2012. Safer asynchronous runtime monitoring using compensations. *Formal Methods Syst. Des.* 41, 3 (2012), 269–294. https://doi.org/10.1007/s10703-012-0142-8

[6] Joshua Ellul. 2020. Towards Configurable and Efficient Runtime Verification of Blockchain Based Smart Contracts at the Virtual Machine Level. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 12478)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 131–145. https://doi.org/10.1007/978-3-030-61467-6_9

[7] Joshua Ellul and Gordon J. Pace. 2018. Runtime Verification of Ethereum Smart Contracts. In *14th European Dependable Computing Conference, EDCC 2018, Iaşi, Romania, September 10-14, 2018*. IEEE Computer Society, 158–163. https://doi.org/10.1109/EDCC.2018.00036

[8] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework For Smart Contracts. *CoRR* abs/1908.09878 (2019). arXiv:1908.09878 http://arxiv.org/abs/1908.09878

[9] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*, Umeshwar Dayal and Irving L. Traiger (Eds.). ACM Press, 249–259. https://doi.org/10.1145/38713.38742

[10] Jim Gray. 1981. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, 144–154.

[11] Giles Reger. 2016. An Overview of MarQ. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10012)*, Yliès Falcone and César Sánchez (Eds.). Springer, 498–503. https://doi.org/10.1007/978-3-319-46982-9_34

[12] Lars Stegeman. 2018. Solitor : runtime verification of smart contracts on the Ethereum network. http://essay.utwente.nl/76902/

[13] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018*. ACM, 9–16. http://ieeexplore.ieee.org/document/8445052

[14] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* (2014).