Distributed Runtime Verification of Metric Temporal Properties for Cross-Chain Protocols

Ritam Ganguly*, Yingjie Xue†, Aaron Jonckheere*, Parker Ljung†, Benjamin Schornstein†,
Borzoo Bonakdarpour*, and Maurice Herlihy†
*Michigan State University {gangulyr, jonckh16, borzoo}@msu.edu
†Brown University {yingjie_xue, parker_ljung, benjamin_schornstein}@brown.edu, mph@cs.brown.edu

***Abstract*—Transactions involving multiple blockchains are implemented by *cross-chain* protocols. These protocols are based on smart contracts, programs that run on blockchains, executed by a network of computers. Verifying the runtime correctness of smart contracts is a problem of compelling practical interest since, smart contracts can automatically transfer ownership of cryptocurrencies, electronic securities, and other valuable assets among untrusting parties. Such verification is challenging since smart contract execution is time sensitive, and the clocks on different blockchains may not be perfectly synchronized. This paper describes a method for runtime monitoring of blockchain executions. First, we propose a generalized runtime verification technique for verifying partially synchronous distributed computations for the metric temporal logic (MTL) by exploiting bounded-skew clock synchronization. Second, we introduce a progression-based formula rewriting scheme for monitoring MTL specifications which employs SMT solving techniques and report experimental results.**

***Index Terms*—Metric Temporal Logic, Partial Synchrony, Distributed Systems, Runtime Verification, Blockchain, Cross-Chain Protocols**

I. Introduction

Blockchain technology [1], [2] is a blockbuster in todays era. It has drawn extensive attention from both industry and academia. With blockchain technology, people can trade in a peer-to-peer manner without mutually trusting each other, removing the necessity of a trusted centralized party. The concept of decentralization appears extremely appealing, and the transparency, anonymity, and persistent storage provided by blockchain make it more attractive. This revolutionary technology has triggered many applications in industry namely, cryptocurrency [3], non-fungible tokens [4], internet of things [5] and health services [6] among others.

Besides the huge success of cryptocurrencies known as blockchain 1.0, especially Bitcoin [2], blockchain 2.0, known as *smart contracts* [7], is also promising in many scenarios. A smart contract is a program running on the blockchain. Its execution is triggered automatically and is enforced by conditions preset in the code. In this way, the transfer of assets can be automated by the rules in the smart contracts, and no human intervention can stop it. A typical smart contract implemented in Ethereum [8], uses *Solidity* [8], which is a Turing-complete language. However, automating the transactions by smart contracts also has its downsides. If the smart contract has bugs and does not do what is expected, then lack of human intervention may lead to massive financial losses. For example, as pointed out by [9], the Parity Multisig Wallet smart contract [10] version 1.5 included a vulnerability which led to the loss of 30 million US dollars. Thus, developing effective techniques to verify the correctness of smart contracts is both urgent and important to protect against possible losses. Furthermore, when a protocol is made up of multiple smart contracts across different blockchains, the correctness of protocols also need to be verified.

This work is sponsored by the United States National Science Foundation (NSF) FMitF Award 2102106.

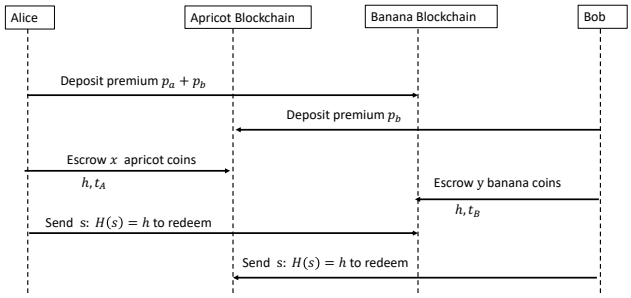


Fig. 1: Hedged Two-party Swap

In this paper, we advocate for a *runtime verification* (RV) approach, to monitor the behavior of a system of blockchains with respect to a set of temporal logic formulas. Applying RV to deal with multiple blockchains can be reduced to *distributed RV*, where a centralized or decentralized monitor observes the behavior of a distributed system in which processes do not share a global clock. Although RV deals with finite executions, the lack of a common global clock prohibits it from having a total unique ordering of events in a distributed setting. Put it another way, the monitor can only form a partial order of event which may result in different verification verdicts. Enumerating all possible partial ordering of events at run time incurs in an exponential blow up, making the approach not scalable. To add to this already complex task, most specifications for verifying blockchain smart contracts, come with a time bound. This means, not only the partial ordering of the events are at play when verifying, but also the actual physical time of occurrence of the events dictates the verification verdict.

In this paper, we propose an effective, sound and complete solution to distributed RV for timed specifications expressed in the *metric temporal logic* (MTL) [11]. To present a high-level

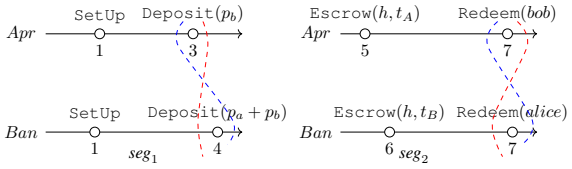


Fig. 2: Progression Example

view of MTL, consider the *two-party swap* protocol [12] shown in Fig 1. Alice and Bob, each in possession of Apricot and Banana blockchain assets respectively, wants to swap their assets between each other without being a victim of a sore loser attack [12][1]. There is a number of requirements that should be followed by the conforming parties to discourage any attack on themselves. We use *metric temporal logic* (MTL) [11] to express such requirements. One such requirement is, where Bob should not be able to redeem his asset before Alice redeems hers within eight time units can be represented by the MTL formula:

$$\varphi_{\mathsf{spec}} = \neg \texttt{Apr.Redeem}(bob)\, \mathcal{U}_{[0,8)} \texttt{Ban.Redeem}(alice).$$

We consider a fault proof central monitor which has the complete view of the system but has no access to a global clock. In order to limit the blow-up of states posed by the absence of a global clock, we make a practical assumption about the presence of a *bounded clock skew* ϵ between the local clocks of every pair of processes, guaranteed by a clock synchronization algorithm (e.g. NTP [13]). This setting is known to be partially synchronous when we do not assume the presence of a global clock and limit the impact of asynchrony within clock drifts. Such an assumption limits the window of partial orders of events only within ϵ time units and significantly reduces the combinatorial blow-up caused by nondeterminism due to concurrency. Existing distributed RV techniques either assume a global clock when working with time sensitive specifications [14], [15] or use untimed specifications when assuming partial synchrony [16], [17].

We introduce an SMT[2]-based *progression-based* formula rewriting technique over distributed computations which takes into consideration the events observed thus far to rewrite the specifications for future extensions. Our monitoring algorithm accounts for all possible orderings of events without explicitly generating them when evaluating MTL formulas. For example, in Fig. 2, we see the events and the time of occurrence in the two blockchains, Apricot(Apr) and Banana(Ban) divided into two segments, seg_1 and seg_2 for computational purposes. Considering maximum clock skew $\epsilon = 2$ and the specification φ_{spec}, at the end of the first segment, we have two possible rewritten formulas for the next segment:

$$\varphi_{\mathsf{spec}_1} = \neg \texttt{Apr.Redeem}(bob) \quad \mathcal{U}_{[0,4)} \texttt{Ban.Redeem}(alice)$$
$$\varphi_{\mathsf{spec}_2} = \neg \texttt{Apr.Redeem}(bob) \quad \mathcal{U}_{[0,3)} \texttt{Ban.Redeem}(alice)$$

This is possible due to the different ordering and different time of occurrence of the events $\texttt{Deposit}(p_b)$ and $\texttt{Deposit}(p_a + p_b)$. In other words, the possible time of occurrence of the event $\texttt{Deposit}(p_b)$ (resp. $\texttt{Deposit}(p_a + p_b)$) is either 2, 3 or 4 (resp. 3, 4, or 5) due to the maximum clock skew of 2. Likewise, at the end of seg_2, we have $\varphi_{\mathsf{spec}_1}$ evaluate to `true` where as $\varphi_{\mathsf{spec}_2}$ evaluate to `false`. This is because, even if we consider the scenario when `Ban.Redeem(alice)` occurs before `Apr.Redeem(bob)`, a possible time of occurrence of `Ban.Redeem(alice)` is 8 (resp. 6) which makes $\varphi_{\mathsf{spec}_2}$ (resp. $\varphi_{\mathsf{spec}_1}$) evaluate to `false` (resp. `true`).

We have fully implemented our technique[3] and report the results of rigorous experiments on monitoring synthetic data, using benchmarks in the tool UPPAAL [18], as well as monitoring correctness, liveness and conformance conditions for smart contracts on blockchains. We put our monitoring algorithm to test, studying the effect of different parameters on the runtime and report on each of them.

Organization: Section II presents the background concepts. Formal statement of our RV problem is discussed in Section III. The formula progression rules and the SMT-based solution are described in Sections IV and V, respectively, while experimental results are analyzed in Section VI. Related work is discussed in Section VII before we make concluding remarks in Section VIII. More details about our case studies can be found in the Appendix, Section IX.

II. Preliminaries

In this section, we present an overview of the distributed computation and the metric temporal logic (MTL).

A. Distributed Computation

A distributed system (e.g., a system of multiple blockchains) can be modeled as a loosely coupled asynchronous system, consisting of n reliable processes (that do not fail), denoted by $\mathcal{P} = \{P_1, P_2, \cdots, P_n\}$. As a system, the processes do not share any memory or have a common global clock. Channels are assumed to be FIFO and lossless. In our model, we represent each local state change and a message activity (send or receive) by an event. Message passing does not change the state of the process and we disregard the content of the message as it is of no use for our monitoring technique. Here, we refer to a global clock which will acts as the "real" time keeper. It is to be noted that the presence of this global clock is just for theoretical reasons and it is not available to any of the individual processes.

We make an assumption about a partially synchronous system. For each process P_i, where $i \in [1, n]$, the local clock can be represented as a monotonically increasing function $c_i : \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$, where $c_i(\mathcal{G})$ is the value of the local

[1] A sore loser attack is a type of attack in cross-blockchain commerce. It occurs when one party decides to halt participation partway through, leaving other parties' assets locked up for a long duration.

[2] *Satisfiability modulo theories* (SMT) is the problem of determining whether a formula involving Boolean expressions comprising of more complex formulas involving real numbers, integers, and/or various data structures is satisfiable.

[3] https://github.com/TART-MSU/rv-mtl-blockc

clock at global time $\mathcal{G}$. Since we are dealing with discrete-time systems, for simplicity and without loss of generality, we represent time with non-negative integers $\mathbb{Z}_{\geq 0}$. For any two processes P_i and P_j, where $i \neq j$, we assume:

$$\forall \mathcal{G} \in \mathbb{Z}_{\geq 0}. \mid c_i(\mathcal{G}) - c_j(\mathcal{G}) \mid < \epsilon,$$

where $\epsilon > 0$ is the maximum clock skew. The value of ϵ is constant and is known to the monitor. This assumption is met by the presence of a clock synchronization algorithm, like NTP [13], to ensure bounded clock skew among all processes. We denote an *event* on process P_i by e^i_σ, where $\sigma = c_i(\mathcal{G})$. That is the local time of occurrence of the event at some global time $\mathcal{G}$.

Definition 1. *A* distributed computation *consisting of n processes is represented by the pair $(\mathcal{E}, \rightsquigarrow)$, where $\mathcal{E}$ is a set of events partially ordered by Lamport's happened-before ($\rightsquigarrow$) relation [19], subject to the partial synchrony assumption:*

- *In every P_i, $1 \leq i \leq n$, all events are totally ordered:*

$$\forall \sigma, \sigma' \in \mathbb{Z}_{\geq 0} : (\sigma < \sigma') \rightarrow (e^i_\sigma \rightsquigarrow e^i_{\sigma'});$$

- *If e is a message sending event in a process and f is the corresponding message receiving event in another process, then we have $e \rightsquigarrow f$;*
- *For any two processes P_i and P_j and two corresponding events $e^i_\sigma, e^j_{\sigma'} \in \mathcal{E}$, if $\sigma + \epsilon < \sigma'$ then, $e^i_\sigma \rightsquigarrow e^j_{\sigma'}$, where ϵ is the maximum clock skew, and*
- *If $e \rightsquigarrow f$ and $f \rightsquigarrow g$, then $e \rightsquigarrow g$.* □

Definition 2. *Given a distributed computation $(\mathcal{E}, \rightsquigarrow)$, a subset of events $\mathcal{C} \subseteq \mathcal{E}$ is said to form a consistent cut if and only if when $\mathcal{C}$ contains an event e, then it should also contain all such events that happened before e. Formally,*

$$\forall e \in \mathcal{E}.(e \in \mathcal{C}) \wedge (f \rightsquigarrow e) \rightarrow f \in \mathcal{C}. \quad \square$$

The frontier of a consistent cut $\mathcal{C}$, denoted by $\mathsf{front}(\mathcal{C})$ is the set of all events that happened last in each process in the cut. That is, $\mathsf{front}(\mathcal{C})$ is a set of e^i_{last} for each $i \in [1, |\mathcal{P}|]$ and $e^i_{last} \in \mathcal{C}$. We denote e^i_{last} as the last event in P_i such that $\forall e^i_\sigma \in \mathcal{C}.(e^i_\sigma \neq e^i_{last}) \rightarrow (e^i_\sigma \rightsquigarrow e^i_{last})$.

B. Metric Temporal Logic (MTL) [20], [21]

Let $\mathbb{I}$ be a set of nonempty intervals over $\mathbb{Z}_{\geq 0}$. We define an interval, $\mathcal{I}$, to be

$$[start, end) \triangleq \{a \in \mathbb{Z}_{\geq 0} \mid start \leq a < end\}$$

where $start \in \mathbb{Z}_{\geq 0}$, $end \in \mathbb{Z}_{\geq 0} \cup \{\infty\}$ and $start < end$. We define AP as the set of all *atomic propositions*, and $\Sigma = 2^{\mathsf{AP}}$ as the set of all possible *states*. A *trace* is represented by a pair which consists of a sequence of states, denoted by $\alpha = s_0 s_1 \cdots$, where $s_i \in \Sigma$ for every $i > 0$ and a sequence of non-negative numbers, denoted by $\bar{\tau} = \tau_0 \tau_1 \cdots$, where $\tau_i \in \mathbb{Z}_{\geq 0}$ for all $i > 0$. We represent the set of all infinite traces by a pair of infinite sets, $(\Sigma^\omega, \mathbb{Z}^\omega_{\geq 0})$. The trace $s_k s_{k+1} \cdots$ (resp. $\tau_k \tau_{k+1}$) is represented by α^k (resp. τ^k). For an infinite trace $\alpha = s_0 s_1 \cdots$ and $\bar{\tau} = \tau_0 \tau_1 \cdots$, $\bar{\tau}$ is an increasing sequence, meaning $\tau_{i+1} \geq \tau_i$, for all $i \geq 0$.

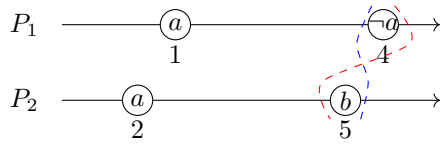


Fig. 3: Different time interleaving of events.

Syntax: The syntax of metric temporal logic (MTL) for infinite traces are defined by the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathcal{U}_{\mathcal{I}} \varphi_2$$

where $p \in \mathsf{AP}$ and $\mathcal{U}_{\mathcal{I}}$ is the 'until' temporal operator with time bound $\mathcal{I}$. We also have $\mathtt{true} = p \vee \neg p$, $\mathtt{false} = \neg \mathtt{true}$, $\varphi_1 \rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\Diamond_{\mathcal{I}} \varphi = \mathtt{true} \mathcal{U}_{\mathcal{I}} \varphi$ ("eventually") and $\Box_{\mathcal{I}} \varphi = \neg(\Diamond_{\mathcal{I}} \neg\varphi)$ ("always"). The set of all MTL formulas is denoted by Φ_{MTL}.

Semantics: The semantics of metric temporal logic (MTL) is defined over $\alpha = s_0 s_1 \cdots$ and $\bar{\tau} = \tau_0 \tau_1 \cdots$ as follows:

$$\begin{array}{lll}
(\alpha, \bar{\tau}, i) \models p & \text{iff} & p \in s_i \\
(\alpha, \bar{\tau}, i) \models \neg\varphi & \text{iff} & (\alpha, \bar{\tau}, i) \not\models \varphi \\
(\alpha, \bar{\tau}, i) \models \varphi_1 \vee \varphi_2 & \text{iff} & (\alpha, \bar{\tau}, i) \models \varphi_1 \vee (\alpha, \bar{\tau}, i) \models \varphi_2 \\
(\alpha, \bar{\tau}, i) \models \varphi_1 \mathcal{U}_{\mathcal{I}} \varphi_2 & \text{iff} & \exists j \geq i.\tau_j - \tau_i \in \mathcal{I} \wedge (\alpha, \bar{\tau}, j) \models \\
 & & \varphi_2 \wedge \forall k \in [i, j), (\alpha, \bar{\tau}, k) \models \varphi_1
\end{array}$$

Also, $(\alpha, \bar{\tau}) \models \varphi$ holds if and only if $(\alpha, \bar{\tau}, 0) \models \varphi$.

In the context of RV, we introduce the notion of finite MTL. The truth values are represented by the set $\mathsf{B}_2 = \{\top, \bot\}$, where $\top$ (resp. $\bot$) represents a formula that is satisfied (resp. violated) given a finite trace. We represent the set of all finite traces by a pair of finite sets, $(\Sigma^*, \mathbb{Z}^*_{\geq 0})$. For a finite trace, $\alpha = s_0 s_1 \cdots s_n$ and $\bar{\tau} = \tau_0 \tau_1 \cdots \tau_n$ the only semantic that needs to be redefined is that of $\mathcal{U}$ ('until') and is as follows:

$$[(\alpha, \bar{\tau}, i) \models_F \varphi_1 \mathcal{U}_{\mathcal{I}} \varphi_2] = \begin{cases} \top & \text{if } \exists j \geq i.\tau_j - \tau_i \in \mathcal{I} \\ & ([\alpha^j \models_F \varphi_2] = \top) \wedge \forall k \in \\ & [i, j) : ([\alpha^k \models_F \varphi_1] = \top) \\ \bot & \text{otherwise.} \end{cases}$$

In order to further illustrate the difference between MTL and finite MTL, consider formula $\varphi = \Diamond_{\mathcal{I}} p$ and a trace $\alpha = s_0 s_1 \cdots s_n$ and $\bar{\tau} = \tau_0 \tau_1 \cdots \tau_n$. We have $[(\alpha, \bar{\tau}) \models_F \varphi] = \top$ if for some $j \in [0, n]$, we have $\tau_j - \tau_0 \in \mathcal{I}$ and $p \in s_i$, otherwise $\bot$. Now, consider formula $\varphi = \Box_{\mathcal{I}} p$. We have $[(\alpha, \bar{\tau}) \models_F \varphi] = \bot$, if for some $j \in [0, n]$, we have $\tau_j - \tau_0 \in \mathcal{I}$ and $p \notin s_i$, otherwise $\top$.

III. Formal Problem Statement

In a partially synchronous system, there are different possible ordering of events and each unique *ordering* of events [22] might evaluate to different RV verdicts. Let $(\mathcal{E}, \rightsquigarrow)$ be a distributed computation. A sequence of consistent cuts is of the form $\mathcal{C}_0 \mathcal{C}_1 \mathcal{C}_2 \cdots$, where for all $i \geq 0$, we have (1) $\mathcal{C}_i \subset \mathcal{C}_{i+1}$ and (2) $|\mathcal{C}_i| + 1 = |\mathcal{C}_{i+1}|$, and (3) $\mathcal{C}_0 = \emptyset$. The set of all sequences of consistent cuts is denoted by $\mathbb{C}$. We note that in our view, the time interval $\mathcal{I}$ in the syntax of MTL represents the physical (global) time $\mathcal{G}$. Thus, when deriving all the possible traces given the distributed computation

$(\mathcal{E}, \rightsquigarrow)$, we account for all different orders in which the events could possibly occur with respect to $\mathcal{G}$. This involves replacing the local time of occurrence of an event, e^i_σ with the set of events $\{e^i_{\sigma'} \mid \sigma' \in [\max\{0, \sigma - \epsilon + 1\}, \sigma + \epsilon)\}$. This is to account for the maximum clock drift that is possible on the local clock of a process when compared to the global clock. For example, given the computation in Fig. 3, a maximum clock skew $\epsilon = 2$ and a MTL formula, $\varphi = a\,\mathcal{U}_{[0,6)}\,b$, one has to consider all possible traces including $(a,1)(a,2)(b,4)(\neg a,5) \models \varphi$ and $(a,1)(a,2)(\neg a,4)(b,5) \not\models \varphi$.

Given a sequence of consistent cuts, it is evident that for all $j > 0$, $|\mathcal{C}_j - \mathcal{C}_{j-1}| = 1$ and event $\mathcal{C}_j - \mathcal{C}_{j-1}$ is the last event that was added onto the cut $\mathcal{C}_j$. To translate monitoring of a distributed system into monitoring a trace, We define a sequence of natural numbers as $\bar{\pi} = \pi_0\pi_1\cdots$, where $\pi_0 = 0$ and for each $j \geq 1$, we have $\pi_j = \sigma$, such that $\mathsf{front}(\mathcal{C}_j) - \mathsf{front}(\mathcal{C}_{j-1}) = \{e^i_\sigma\}$. To maintain time monotonicity, we only consider sequences where for all $i \geq 0$, $\pi_{i+1} \geq \pi_i$.

The set of all traces that can be formed from $(\mathcal{E}, \rightsquigarrow)$ is defined as:

$$\mathsf{Tr}(\mathcal{E}, \rightsquigarrow) = \left\{ \mathsf{front}(\mathcal{C}_0)\mathsf{front}(\mathcal{C}_1)\cdots \mid \mathcal{C}_0\mathcal{C}_1\cdots \in \mathbb{C} \right\}$$

In the sequel, we assume that every sequence α of frontiers in $\mathsf{Tr}(\mathcal{E}, \rightsquigarrow)$ is associated with a sequence $\bar{\pi}$. Thus, to comply with the semantics of MTL, we refer to the elements of $\mathsf{Tr}(\mathcal{E}, \rightsquigarrow)$ by pairs $(\alpha, \bar{\pi})$. Thus, we evaluate an MTL formula φ with respect to a computation $(\mathcal{E}, \rightsquigarrow)$ as follows:

$$[(\mathcal{E}, \rightsquigarrow) \models_F \varphi] = \left\{ (\alpha, \bar{\pi}, 0) \models_F \varphi \mid (\alpha, \bar{\pi}) \in \mathsf{Tr}(\mathcal{E}, \rightsquigarrow) \right\}$$

This boils down to having a set of verdicts, since a distributed computation may involve several traces and each trace might evaluate to a different verdict.

Overall idea of our solution: To solve the above problem (evaluating all possible verdicts), we propose a monitoring approach based on formula-rewriting (Section IV) and SMT solving (Section V). Our approach involves iteratively (1) chopping a distributed computation into a sequence of smaller segments to reduce the problem size and (2) progress the MTL formula for each segment for the next segment, which results in a new MTL formula by invoking an SMT solver. Since each computation/segment corresponds to a set of possible traces due to partial synchrony, each invocation of the SMT solver may result in a different verdict.

IV. Formula Progression for MTL

We start describing our solution by explaining the formula progression technique.

Definition 3. *A* progression function *is of the form* $\mathsf{Pr} : \Sigma^* \times \mathbb{Z}^*_{\geq 0} \times \Phi_{MTL} \to \Phi_{MTL}$ *and is defined for all finite traces* $(\alpha, \bar{\tau}) \in (\Sigma^*, \mathbb{Z}^*_{\geq 0})$*, infinite traces* $(\alpha', \bar{\tau}') \in (\Sigma^\omega, \mathbb{Z}^\omega_{\geq 0})$ *and MTL formulas* $\varphi \in \Phi_{MTL}$*, such that* $(\alpha.\alpha', \bar{\tau}.\bar{\tau}') \models \varphi$ *if and only if* $(\alpha', \bar{\tau}') \models \mathsf{Pr}(\alpha, \bar{\tau}, \varphi)$. □

Compared to the classic formula rewriting technique in [23], here the function Pr takes a finite trace as input, while the algorithm in [23] rewrites the formula after every observed state. When monitoring a partially synchronous distributed system, multiple verdicts are possible as a result of no unique ordering of events, as a result the classical state-by-state formula rewriting technique is of little use. The motivation of our approach comes from the fact that for computation reasons, we chop the computation into smaller segments and the verification of each segment is done through an SMT query. A state-by-state approach would incur in a huge number of SMT queries being generated.

Let $\mathcal{I} = [start, end)$ denote an interval. By $\mathcal{I} - \tau$, we mean the interval $\mathcal{I}' = [start', end')$, where $start' = \max\{0, start - \tau\}$ and $end' = \max\{0, end - \tau\}$. Also, for two time instances τ_i and τ_0, we let InInt(i) return `true` or `false` depending upon whether $\tau_i - \tau_0 \in \mathcal{I}$.

Progressing atomic propositions. For an MTL formula of the form $\varphi = p$, where $p \in \mathsf{AP}$, the result depends on whether or not $p \in \alpha(0)$. This marks as our base case for the other temporal and logical operators:

$$\mathsf{Pr}(\alpha, \bar{\tau}, \varphi) = \begin{cases} \texttt{true} & \text{if } \ p \in \alpha(0) \\ \texttt{false} & \text{if } \ p \notin \alpha(0) \end{cases}$$

Progressing negation. For an MTL formula of the form $\varphi = \neg\phi$, we have:

$$\mathsf{Pr}(\alpha, \bar{\tau}, \varphi) = \neg\mathsf{Pr}(\alpha, \bar{\tau}, \phi).$$

Progressing disjunction. Let $\varphi = \varphi_1 \vee \varphi_2$. Apart from the trivial cases, the result of progression of $\varphi_1 \vee \varphi_2$ is based on progression of φ_1 and/or progression of φ_2:

$$\mathsf{Pr}(\alpha, \bar{\tau}, \varphi) = \begin{cases} \texttt{true} & \text{if } \ \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_1) = \texttt{true} \vee \\ & \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_2) = \texttt{true} \\ \texttt{false} & \text{if } \ \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_1) = \texttt{false} \wedge \\ & \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_2) = \texttt{false} \\ \varphi'_2 & \text{if } \ \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_1) = \texttt{false} \wedge \\ & \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_2) = \varphi'_2 \\ \varphi'_1 & \text{if } \ \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_2) = \texttt{false} \wedge \\ & \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_1) = \varphi'_1 \\ \varphi'_1 \vee \varphi'_2 & \text{if } \ \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_1) = \varphi'_1 \wedge \\ & \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_2) = \varphi'_2 \end{cases}$$

Always and eventually operators. As shown in Algorithms 1 and 2, the progression for 'always', $(\Box_{\mathcal{I}}\,\varphi)$ and 'eventually', $(\Diamond_{\mathcal{I}}\,\varphi)$ depends on the value of InInt(i) and the progression of the inner formula φ. In Algorithms 1 and 2, we divide the algorithm into three cases: (1) line 4, corresponds to if $\mathcal{I}$ is within the sequence $\bar{\tau}$; (2) line 6, corresponds to where $\mathcal{I}$ starts in the current trace but its end is beyond the boundary of the sequence $\bar{\tau}$, and (3) line 9, corresponds to if the entire interval $\mathcal{I}$ is beyond the boundary of sequence $\bar{\tau}$. In Algorithm 1, we are only concerned about the progression of φ on the suffix $(\alpha^i, \bar{\tau}^i)$ if InInt(i) = `true`. In case, InInt(i) = `false` the consequent drops and the entire condition equates to `true`. In other words, equating over all $i \in [0, |\alpha|]$, we are only

Algorithm 1 Always

1: **function** $\mathsf{Pr}(\alpha, \bar{\tau}, \Box_{\mathcal{I}}\, \varphi)$
2: **if** $\mathcal{I}_{start} \leq \tau_{|\alpha|} - \tau_0$ **then**
3: **if** $\mathcal{I}_{end} \leq \tau_{|\alpha|} - \tau_0$ **then**
4: **return** $\bigwedge_{i\in[0,|\alpha|]} \left(\mathsf{InInt(i)} \rightarrow \mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi)\right)$
5: **else**
6: **return** $\bigwedge_{i\in[0,|\alpha|]} \left(\mathsf{InInt(i)} \rightarrow \mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi)\right) \wedge \Box_{[\mathcal{I}-(\tau_{|\alpha|}-\tau_0))}\, \varphi$
7: **end if**
8: **else**
9: **return** $\Box_{[\mathcal{I}-(\tau_{|\alpha|}-\tau_0))}\, \varphi$
10: **end if**
11: **end function**

Algorithm 2 Eventually

1: **function** $\mathsf{Pr}(\alpha, \bar{\tau}, \Diamond_{\mathcal{I}}\, \varphi)$
2: **if** $\mathcal{I}_{start} \leq \tau_{|\alpha|} - \tau_0$ **then**
3: **if** $\mathcal{I}_{end} \leq \tau_{|\alpha|} - \tau_0$ **then**
4: **return** $\bigvee_{i\in[0,|\alpha|]} \left(\mathsf{InInt(i)} \wedge \mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi)\right)$
5: **else**
6: **return** $\bigvee_{i\in[0,|\alpha|]} \left(\mathsf{InInt(i)} \wedge \mathsf{Pr}(\alpha^i, \bar{\tau}^i, \phi)\right) \vee \Diamond_{[\mathcal{I}-(\tau_{|\alpha|}-\tau_0))}\, \varphi$
7: **end if**
8: **else**
9: **return** $\Diamond_{[\mathcal{I}-(\tau_{|\alpha|}-\tau_0))}\, \varphi$
10: **end if**
11: **end function**

Algorithm 3 Until

1: **function** $\mathsf{Pr}(\alpha, \bar{\tau}, \varphi_1\, \mathcal{U}_{\mathcal{I}} \varphi_2)$
2: **if** $\mathcal{I}_{start} \leq \tau_{|\alpha|} - \tau_0$ **then**
3: **if** $\mathcal{I}_{end} \leq \tau_{|\alpha|} - \tau_0$ **then**
4: **return** $\left(\bigwedge_{i\in[0,|\alpha|]} \left((\tau_i < \mathcal{I}_{start} + \tau_0) \rightarrow \mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi_1)\right)\right) \wedge \left(\bigvee_{j\in[0,|\alpha|]} \left(\mathsf{InInt(j)} \wedge \mathsf{Pr}(\alpha, \bar{\tau}, \Box_{[0,\tau_j-\tau_0)}\, \varphi_1) \wedge \mathsf{Pr}(\alpha^j, \bar{\tau}^j, \varphi_2)\right)\right)$
5: **else**
6: **return** $\left(\bigwedge_{i\in[0,|\alpha|]} \left((\tau_i < \mathcal{I}_{start} + \tau_0) \rightarrow \mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi_1)\right)\right) \wedge \left(\bigvee_{j\in[0,|\alpha|]} \left(\mathsf{InInt(j)} \wedge \mathsf{Pr}(\alpha, \bar{\tau}, \Box_{[0,\tau_j-\tau_0)}\, \varphi_1) \wedge \mathsf{Pr}(\alpha^j, \bar{\tau}^j, \varphi_2)\right) \vee \varphi_1\, \mathcal{U}_{(\mathcal{I}-(\tau_{|\alpha|}-\tau_0)\varphi_2}\right)$
7: **end if**
8: **else**
9: **return** $\left(\bigwedge_{i\in[0,|\alpha|]} \mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi_1)\right) \wedge \varphi_1\, \mathcal{U}_{(\mathcal{I}-(\tau_{|\alpha|}-\tau_0)\varphi_2}$
10: **end if**
11: **end function**

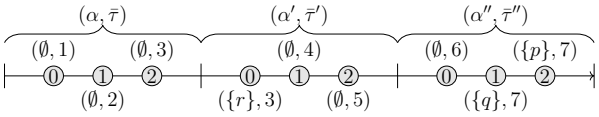


Fig. 4: A trace example divided into three segments

left with conjunction of $\mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi)$ where $\mathsf{InInt(i)} = \texttt{true}$. In addition to this, we add the initial formula with updated interval for the next trace. Similarly, in Algorithm 2, equating over all $i \in [0, |\alpha|]$, if $\mathsf{InInt(i)} = \texttt{false}$ the corresponding $\mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi)$ is disregarded and the final formula is a disjunction of $\mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi)$ with $\mathsf{InInt(i)} = \texttt{true}$.

Progressing the until operator. Let the formula be of the form $\varphi_1\, \mathcal{U}_{\mathcal{I}} \varphi_2$. According to the semantics of until, φ_1 should be evaluated to true in all states leading up to some $i \in \mathcal{I}$, where φ_2 evaluates to true. We start by progressing φ_1 (resp. φ_2) as $\Box_{[0,\tau_i-\tau_0)}\, \varphi_1$ (resp. $\Diamond_{[\tau_i,\tau_i+1)}\, \varphi_2$) for some $i \in \mathcal{I}$. Since, we are only verifying the sub-formula, $\Diamond_{[\tau_i,\tau_i+1)}\, \varphi_2$, on the trace sequence $(\alpha, \bar{\tau})$, it is equivalent to verifying the sub-formula $\Diamond_{[0,1)}\, \varphi_2 \equiv \varphi_2$ over the trace sequence $(\alpha^i, \bar{\tau}^i)$. Similar to Algorithms 1 and 2, in Algorithm 3 we need to consider three cases. In lines 4, 6 and 9, following the semantics of until operator, we make sure for all $i \in [0, |\alpha|]$, if $\tau_i < \mathcal{I}_{start} + \tau_0$, φ_1 is satisfied in the suffix $(\alpha^i, \bar{\tau}^i)$. In addition to this there should be some $j \in [0, |\alpha|]$ for which if $\mathsf{InInt(j)} = \texttt{true}$, then the trace satisfies the sub-formula $\Box_{[0,\tau_j-\tau_0)}\, \varphi_1$ and $\Diamond_{[\tau_j,\tau_j+1)}\, \varphi_2$). In lines 6 and 9, we also accommodate for future traces satisfying the formula $\varphi_1\, \mathcal{U}_{\mathcal{I}} \varphi_2$ with updated intervals.

Example: In Fig. 4, the time line shows propositions and their time of occurrence, for formula $\Diamond_{[0,6)}\, r \rightarrow (\neg p\, \mathcal{U}_{[2,9)} q)$. The entire computation has been divided into 3 segments, $(\alpha, \bar{\tau})$, $(\alpha', \bar{\tau}')$, and $(\alpha'', \bar{\tau}'')$ and each state has been represented by (s, τ):

- We start with segment $(\alpha, \bar{\tau})$. First we evaluate $\Diamond_{[0,6)}\, r$, which requires evaluating $\mathsf{Pr}(\alpha^i, \bar{\tau}^i, r)$ for $i \in \{0, 1, 2\}$, all of which returns the verdict $\texttt{false}$ and there by rewriting the sub-formula as $\Diamond_{[0,4)}\, r$. Next, to evaluate the sub-formula $\neg p\, \mathcal{U}_{[2,9)} q$, we need to evaluate (1) $\mathsf{Pr}(\alpha^i, \bar{\tau}^i, \neg p)$ for $i \in \{0, 1\}$ since $\tau_i - \tau_0 < 2$ and both evaluates to $\texttt{true}$, (2) $\mathsf{Pr}(\alpha, \bar{\tau}, \Box_{[0,2)}\, \neg p)$ which also evaluates to $\texttt{true}$ and (3) $\mathsf{Pr}(\alpha^2, \bar{\tau}^2, q)$ which evaluates as $\texttt{false}$. Thereby, the rewritten formula after observing $(\alpha, \bar{\tau})$ is $\Diamond_{[0,3)}\, r \rightarrow (\neg p\, \mathcal{U}_{[0,6)} q)$.
- Similarly, we evaluate the formula now with respect to $(\alpha', \bar{\tau}')$, which makes the sub-formula $\Diamond_{[0,3)}\, r$ evaluate to $\texttt{true}$ at $\tau = 3$ and the sub-formula $\neg p\, \mathcal{U}_{[0,6)} q$ (there is no such $i \in \{0, 1, 2\}$ where $\tau_i - \tau_0 < 0$ and for all $j \in \{0, 1, 2\}$, $\mathsf{Pr}(\alpha'^j, \bar{\tau}'^j, q) = \texttt{false}$) is rewritten as $\neg p\, \mathcal{U}_{[0,4)} q$.
- In $(\alpha'', \bar{\tau}'')$, for $j = 1$, $\mathsf{Pr}(\alpha'', \bar{\tau}'', \Box_{[0,2)}\, \neg p) = \texttt{true}$ and $\mathsf{Pr}(\alpha''^j, \bar{\tau}''^j, q) = \texttt{true}$, and thereby rewriting the entire formula as $\texttt{true}$.

V. SMT-based Solution

A. SMT Entities

SMT entities represent the variables used to represent the distributed computation. After we have the verdicts for each of the individual sub-formulas, we use the progression laws

discussed in Section IV to construct the formula for the future computations.

Distributed Computation We represent a distributed computation $(\mathcal{E}, \rightsquigarrow)$ by a function $f : \mathcal{E} \to \{0, 1, \ldots, |\mathcal{E}| - 1\}$. To represent the happen-before relation, we define a $\mathcal{E} \times \mathcal{E}$ matrix called hbSet where $\mathsf{hbSet}[e^i_\sigma][e^j_{\sigma'}] = 1$ represents $e^i_\sigma \rightsquigarrow e^j_{\sigma'}$ for $e^i_\sigma, e^j_{\sigma'} \in \mathcal{E}$. Also, if $|\sigma - \sigma'| \geq \epsilon$ then $\mathsf{hbSet}[e^i_\sigma][e^j_{\sigma'}] = 1$, else $\mathsf{hbSet}[e^i_\sigma][e^j_{\sigma'}] = 0$. This is done in the pre-processing phase of the algorithm and in the rest of the paper, we represent events by the set $\mathcal{E}$ and a happen-before relation by $\rightsquigarrow$ for simplicity.

In order to represent the possible time of occurrence of an event, we define a function $\delta : \mathcal{E} \to \mathbb{Z}_{\geq 0}$, where

$$\forall e^i_\sigma \in \mathcal{E}. \exists \sigma' \in [\max\{0, \sigma - \epsilon + 1\}, \sigma + \epsilon - 1]. \delta(e^i_\sigma) = \sigma'$$

To connect events, $\mathcal{E}$, and propositions, AP, on which the MTL formula φ is constructed, we define a boolean function $\mu : \mathsf{AP} \times \mathcal{E} \to \{\mathtt{true}, \mathtt{false}\}$. For formulas involving non-boolean variables (e.g., $x_1 + x_2 \leq 7$), we can update the function μ accordingly. We represent a sequence of consistent cuts that start from $\{\}$ and end in $\mathcal{E}$, we introduce an *uninterpreted function* $\rho : \mathbb{Z}_{\geq 0} \to 2^{\mathcal{E}}$ to reach a verdict, given it satisfies all the constrains explained in V-B. Lastly, to represent the sequence of time associated with the sequence of consistent cuts, we introduce a function $\tau : \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$.

B. SMT Constraints

Once we have the necessary SMT entities, we move onto including the constraints for both generating a sequence of consecutive cuts and also representing the MTL formula as a SMT constraint.

Consistent cut constraints over ρ: In order to make sure the sequence of cuts represented by the uninterpreted function ρ, is a sequence of consistent cuts, i.e., they follow the happen-before relations between events in the distributed system:

$$\forall i \in [0, |\mathcal{E}|]. \forall e, e' \in \mathcal{E}. \Big((e' \rightsquigarrow e) \wedge \big(e \in \rho(i)\big) \Big) \to \big(e' \in \rho(i)\big)$$

Next, we make sure that in the sequence of consistent cuts, the number of events present in a consistent cut is one more than the number of events that were present in the consistent cut before it:

$$\forall i \in [0, |\mathcal{E}|). \mid \rho(i+1) \mid = \mid \rho(i) \mid +1$$

Next, we make sure than in the sequence of consistent cuts, each consistent cut includes all the events that were present in the consistent cut before it, i.e, it is a subset of the consistent cut prior in the sequence.

$$\forall i \in [0, |\mathcal{E}|]. \rho(i) \subset \rho(i+1)$$

The sequence of consistent cuts starts from $\{\}$ and ends at $\mathcal{E}$.

$$\rho(0) = \emptyset;\ \rho(|\mathcal{E}|) = \mathcal{E}$$

The sequence of time reflects the time of occurrence of the event that has just been added to the sequence of consistent cut:

$$\forall i \geq 1. \tau(i) = \delta(e^i_\sigma), \text{ such that } \rho(i) - \rho(i-1) = \{e^i_\sigma\}$$

And finally, we make sure the monotonicity of time is maintained in the sequence of time

$$\forall i \in [0, |\mathcal{E}|). \tau(i+1) \geq \tau(i)$$

Constraints for MTL formulas over ρ: These constraints will make sure that ρ will not only represent a valid sequence of consistent cuts but also makes sure that the sequence of consistent cuts satisfy the MTL formula. As is evident, a distributed computation can often yield two contradicting evaluation. Thus, we need to check for both satisfaction and violation for all the sub-formulas in the MTL formula provided. Note that monitoring any MTL formula using our progression rules will result in monitoring sub-formulas which are atomic propositions, eventually and globally temporal operators. Below we mention the SMT constrain for each of the different sub-formula. Violation (resp. satisfaction) for atomic proposition and eventually (resp. globally) constrain will be the negation of the one mentioned.

$$\varphi = \mathsf{p} \qquad \bigvee_{e \in \mathsf{front}(\rho(0))} \mu[\mathsf{p}, e] = \mathtt{true}, \text{for } \mathsf{p} \in \mathsf{AP}$$

(satisfaction, i.e., $\top$)

$$\varphi = \Box_{\mathcal{I}} \varphi \qquad \exists i \in [0, |\mathcal{E}|]. \tau(i) - \tau(0) \in \mathcal{I} \wedge \rho(i) \not\models \varphi$$

(violation, i.e., $\bot$)

$$\varphi = \Diamond_{\mathcal{I}} \varphi \qquad \exists i \in [0, |\mathcal{E}|]. \tau(i) - \tau(0) \in \mathcal{I} \wedge \rho(i) \models \varphi$$

(satisfaction, i.e., $\top$)

A satisfiable SMT instance denotes that the uninterpreted function was not only able to generate a valid sequence of consistent cuts but also that the sequence satisfies the MTL formula given the computation. This result is then fed to the progression cases to generate the final verdict.

C. Segmentation and Parallelization of Distributed Computation

We know that predicate detection, let alone runtime verification, is NP-complete [24] in the size of the system (number of processes). This complexity grows to higher classes when working with nested temporal operators. To make the problem computationally viable, we aim to chop the computation, $(\mathcal{E}, \rightsquigarrow)$ into g segments, $(seg_1, \rightsquigarrow), (seg_2, \rightsquigarrow), \cdots, (seg_g, \rightsquigarrow)$. This involves creating small SMT-instances for each of the segments which improves the runtime of the overall problem. In a computation of length l, if we were to chop it into g segments, each segment would of the length $\frac{l}{g} + \epsilon$ and the set of events included in it can be given by:

$$seg_j = \Big\{ e^i_\sigma \mid \sigma \in \Big[max\big(0, \frac{(j-1) \times l}{g} - \epsilon\big), \frac{j \times l}{g} \Big] \wedge i \in [1, \mid \mathcal{P} \mid] \Big\}$$

Note that monitoring of a segment should include the events that happened within ϵ time of the segment actually starting since it might include events that are concurrent with some other events in the system not accounted for in the previous segment.

VI. Case Study and Evaluation

In this section, we analyze our SMT-based solution. We note that we are not concerned about data collections, data transfer, etc, as given a distributed setting, the runtime of the actual SMT encoding will be the most dominating aspect of the monitoring process. We evaluate our proposed solution using traces collected from benchmarks of the tool UPPAAL [18][4] models (Section VI-A) and a case study involving smart contracts over multiple blockchains (Section VI-B).

A. UPPAAL Benchmarks

1) Setup: We base our synthetic experiments on 3 different UPPAAL benchmark models described in [25]. *The Train Gate* models a railway control system which controls access to a bridge. The bridge is controlled by a gate/operator and can be accessed by one train at a time. We monitor two properties:

$$\varphi_1 = (\bigwedge_{i \in \mathcal{P}} \neg \texttt{Train[i].Cross})\ \mathcal{U}\ \texttt{Train[1].Cross}$$

$$\varphi_2 = \bigwedge_{i \in \mathcal{P}} \Box\big(\texttt{Train[i].Appr} \rightarrow \Diamond(\texttt{Gate.Occ}\ \mathcal{U}\ \texttt{Train[i].Cross})\big)$$

where $\mathcal{P}$ is the set of trains.

Fischer's Protocol is a mutual exclusion protocol for n processes. We verify first, that no two process (`P`) enter the critical section (`cs`) at the same time and second, all request (`req`) should be followed by the processes that are able to access the critical section within some time.

$$\varphi_3 = \Box(\sum_{i \in \mathcal{P}} \texttt{P[i].cs} \leq 1)$$

$$\varphi_4 = \Box(\bigwedge_{i \in \mathcal{P}} \texttt{P[i].req} \rightarrow \Diamond_{\mathcal{I}}\, \texttt{P[i].cs})$$

The Gossiping People is a model consisting of n people who wish to share their secret with each other. We monitor first, that each `Person` gets to know about everyone else's `secret` within some time bound and second, each `Person` has `secrets` to share infinitely often.

$$\varphi_5 = \Diamond_{\mathcal{I}}(\bigwedge_{i,j \in \mathcal{P}} (i \neq j) \rightarrow \texttt{Person[i].secret[j]})$$

$$\varphi_6 = \bigwedge_{i \in \mathcal{P}} \Box(\Diamond_{\mathcal{I}}\, \texttt{Person[i].secrets})$$

Each experiment involves two steps: (1) distributed computation/trace generation and (2) trace verification. For each UPPAAL model, we consider each pair of consecutive events are $0.1s$ apart, i.e., there are 10 events per second per process. For our verification step, our monitoring algorithm executes on the generated computation and verifies it against an MTL specification. We consider the following parameters (1) primary which includes time synchronization constant (ϵ), (2) MTL formula under monitoring, (3) number of segments (g), (3) computation length (l), (4) number of processes in the system ($\mathcal{P}$), and (5) event rate. We study the runtime of our monitoring algorithm against each of these parameters. We use a machine with $2x$ Intel Xeon Platinum 8180 (2.5 Ghz) processor, 768 GB of RAM, 112 vcores with gcc version 9.3.1.

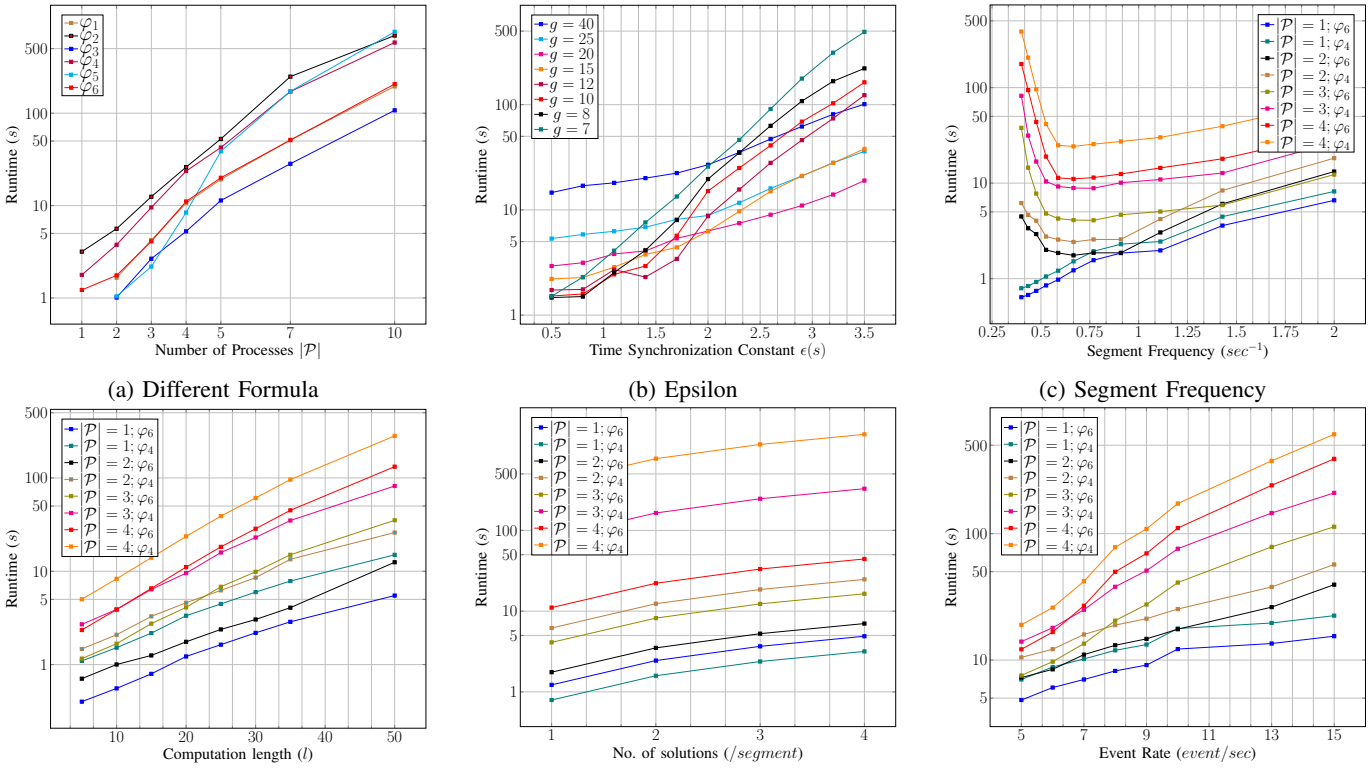
2) Analysis: We study each of the parameters individually and analyze how it effects the runtime of our monitoring approach. All results correspond to $\epsilon = 15ms$, $|\mathcal{P}| = 2$, $g = 15$, $l = 2sec$, an event rate of $10events/sec$ and φ_4 as the MTL specification unless mentioned otherwise. We vary the number of processes in the system from 2 to 4, since in most cross-chain transactions the number of blockchains involved is small.

Impact of different formula. Fig. 5a shows that runtime of the monitor depends on two factors: the number of sub-formulas and the depth of nested temporal operators. Comparing φ_3 and φ_6, both of which consists of the same number of predicates but since φ_6 has recursive temporal operators, it takes more time to verify and the runtime is comparable to φ_1, which consists of two sub-formulas. This is because verification of the inner temporal formula often requires observing states in the next segment in order to come to the final verdict. This accounts for more runtime for the monitor.

Impact of epsilon. Increasing the value of time synchronization constant (ϵ), increases the possible number of concurrent events that needs to be considered. This increases the complexity of verifying the computation and there-by increasing the runtime of the algorithm. In addition to this, higher values of ϵ also correspond to more number of possible traces that are possible and should be taken into consideration. We observe that the runtime increases exponentially with increasing the value of time synchronization constant in Fig. 5b. An interesting observation is that, with longer segment length, the runtime increases at a higher rate than with shorter segment length. This is because with longer segment length and higher ϵ, it equates to a larger number of possible traces that the monitoring algorithm needs to take into consideration. This increases the overall runtime of the verification algorithm by a considerable amount and at a higher pace.

Impact of segment frequency. Increasing the segment frequency makes the length of each segment lower and thus verifying each segment involves a lower number of events. We observe the effect of segment frequency on the runtime of our verification algorithm in Fig. 5c. With increasing the segment frequency, the runtime decreases unless it reaches a certain value (here it is ≈ 0.6) after which the benefit of working with a lower number of events is overcast by the time required to setup each SMT instances. Working with higher

[4]UPPAAL is a model checker for a network of timed automata. The toolset is accompanied by a set of benchmarks for real-time systems. Here, we assume that the components of the network are partially synchronized.



(a) Different Formula (b) Epsilon (c) Segment Frequency

(d) Computation Length (e) Number of Process (f) Event Rate

Fig. 5: Impact of different parameters on synthetic data

number of segments equates to solving more number of SMT problem for the same computation length. Setting up the SMT problem requires a considerable amount of time which is seen by the slight increase in runtime for higher values of segment frequency.

Impact of computation length. As it can be inferred from the previous results, the runtime of our verification algorithm is majorly dictated by the number of events in the computation. Thus, when working with a longer computation, keeping the maximum clock skew and the number of segments constant, we should see a longer verification time as well. Results in Fig. 5d supports the above claim.

Impact of number truth values per segment. In order to take into consideration all possible truth values of a computation, we execute the SMT problem multiple times, with the verdict of all previous executions being added to the SMT problem such that no two verdict is repeated. Here in Fig. 5e we see that the runtime is linearly effected by increasing number of distinct verdicts. This is because, the complexity of the problem that the SMT is trying to solve does not change when trying to evaluate to a different solution.

Impact of event-rate. Increasing the event rate involves more number of events that needs to be processed by our verification algorithm per segment and thereby increasing the runtime at an exponential rate as seen in Fig. 5f. We also observe that with higher number of processes, the rate at which the runtime of our algorithm increases is higher for the same increase in event rate.

B. Blockchain

1) Setup: We implemented the following cross-chain protocols from [12]: two-party swap, multi-party swap, and auction. The protocols are written as smart contracts in Solidity and tested using Ganache, a tool that creates mocked Ethereum blockchains. Using a single mocked chain, we mimicked cross-chain protocols via several (discrete) tokens and smart contracts, which do not communicate with each other.

We use the hedged two-party swap example from [12] to describe our experiments. The implementation of the other two protocols are similar. Suppose Alice would like to exchange her apricot tokens with Bob's banana tokens, using the hedged two-party swap protocol shown in Fig. 1. This protocol provides protection for parties compared to a standard two-party swap protocol [26], in that if one party locks their assets to exchange which is refunded later, this party gets a premium as compensation for locking their assets. The protocol consists of six steps to be executed by Alice and Bob in turn. In our example, we let the amount of tokens they are exchanging be 100 ERC20 tokens and the premium p_b be 1 token and $p_a + p_b$ be 2 tokens. We deploy two contracts on both apricot blockchain(the contract is denoted as $ApricotSwap$) and banana blockchain (denoted as $BananaSwap$) by mimicking the two blockchains on Ethereum. Denote the time that they reach an agreement of the swap as $startTime$. Δ is the maximum

time for parties to observe the state change of contracts by others and take a step to make changes on contracts. In our experiment, $\Delta = 500$ milliseconds. By the definition of the protocol, the execution should be:

- Step 1. Alice deposits 2 tokens as premium in $BananaSwap$ before Δ elapses after $startTime$.
- Step 2. Bob should deposit 1 token as premium in $ApricotSwap$ before 2Δ elapses after $startTime$.
- Step 3. Alice escrows her 100 ERC20 tokens to $ApricotSwap$ before 3Δ elapses after $startTime$.
- Step 4. Bob escrows her 100 ERC20 tokens to $BananaSwap$ before 4Δ elapses after $startTime$.
- Step 5. Alice sends the preimage of the hashlock to $BananaSwap$ to redeem Bob's 100 tokens before 5Δ elapses after $startTime$. Premium is refunded.
- Step 6. Bob sends the preimage of the hashlock to $ApricotSwap$ to redeem Alice's 100 tokens before 6Δ elapses after $startTime$. Premium is refunded.

If all parties are conforming, the protocol is executed as above. Otherwise, some asset refund and premium redeem events is triggered to resolve the case where some party deviates. To avoid distraction, we do not provide details here.

Each smart contract provides functions to let parties deposit premiums `DepositPremium()`, escrow an asset `EscrowAsset()`, send a secret to redeem assets `RedeemAsset()`, refund the asset if it is not redeemed after timeout, `RefundAsset()`, and counterparts for premiums `RedeemPremium()` and `RefundPremium()`. Whenever a function is called successfully (meaning the transaction sent to the blockchain is included in a block), the blockchain emits an event that we then capture and log. The event interface is provided by the Solidity language. For example, when a party successfully calls `DepositPremium()`, the `PremiumDeposited` event emits on the blockchain. We then capture and log this event, allowing us to view the values of `PremiumDeposited`'s declared fields: the time when it emits, the party that initiated `DepositPremium()`, and the amount of premium sent. Those values are later used in the monitor to check against the specification.

2) Log Generation and Monitoring: Our tests simulates different executions of the protocols and generated 1024, 4096, and 3888 different sets of logs for the aforementioned protocols, respectively. We again use the hedged two-party swap as an example to show how we generate different logs to simulate different execution of the protocol. On each contract, we enforce the order of those steps to be executed. For example, step 3 `EscrowAsset()` on the $ApricotSwap$ cannot be executed before Step 1 is taken, i.e. the premium is deposited. This enforcement in the contract restricts the number of possible different states in the contract. Assume we use a binary indicator to denote whether a step is attempted by the corresponding party. 1 denotes a step is attempted, and 0 denotes this step is skipped. If the previous step is skipped, then the later step does not need to be attempted since it will be rejected by the contract. We use an array to denote whether each step is taken for each contract. On each contract, the different executions of those steps can be [1,1,1] meaning all steps are attempted, or [1,1,0] meaning the last step is skipped, and so on. Each chain has 4 different executions. We take the Cartesian product of arrays of two contracts to simulate different combinations of executions on two contracts. Furthermore, if a step is attempted, we also simulate whether the step is taken late, or in time. Thus we have 2^6 possibilities of those 6 steps. In summary, we succeeded generating $4 \cdot 4 \cdot 2^6 = 1024$ different logs.

In our testing, after deploying the two contracts, we iterate over a 2D array of size 1024×12, and each time takes one possible execution denoted as an array length of 12 to simulate the behavior of participants. For example, $[1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]$ stands for the first step is attempted but it is late, and the steps after second step are all attempted in time. Indexed from 0, the even index denotes if a step is attempted or not and the odd index denotes the former step is attempted in time or late. By the indicator given by the array, we let parties attempt to call a function of the contract or just skip. In this way, we produce 1024 different logs containing the events emitted in each iteration.

We check the policies mentioned in [12]: liveness, safety, and ability to hedge against sore loser attacks. *Liveness* means that Alice should deposit her premium on the banana blockchain within Δ from when the swap started($\Diamond_{[0,\Delta)}$ `ban.premium_deposited(alice)`) and then Bob should deposit his premiums, and then they escrow their assets to exchange, redeem their assets (i.e. the assets are swapped), and the premiums are refunded. In our testing, we always call a function to settle all assets in the contract if the asset transfer is triggered by timeout. Thus, in the specification, we also check all assets are settled:

$$\begin{aligned}\varphi_{\text{liveness}} = {} & \Diamond_{[0,\Delta)}\,\texttt{ban.premium_deposited(alice)} \wedge \\ & \Diamond_{[0,2\Delta)}\,\texttt{apr.premium_deposited(bob)} \wedge \\ & \Diamond_{[0,3\Delta)}\,\texttt{apr.asset_escrowed(alice)} \wedge \\ & \Diamond_{[0,4\Delta)}\,\texttt{ban.asset_escrowed(bob)} \wedge \\ & \Diamond_{[0,5\Delta)}\,\texttt{ban.asset_redeemed(alice)} \wedge \\ & \Diamond_{[0,6\Delta)}\,\texttt{apr.asset_redeemed(bob)} \wedge \\ & \Diamond_{[0,5\Delta)}\,\texttt{ban.premium_refunded(alice)} \wedge \\ & \Diamond_{[0,6\Delta)}\,\texttt{apr.premium_refunded(bob)} \wedge \\ & \Diamond_{[6\Delta,\infty)}\,\texttt{apr.all_asset_settled(any)} \wedge \\ & \Diamond_{[5\Delta,\infty)}\,\texttt{ban.all_asset_settled(any)}\end{aligned}$$

Safety is provided only for conforming parties, since if one party is deviating and behaving unreasonably, it is out of the scope of the protocol to protect them. Alice should always deposit her premium first to start the execution of the protocol($\Diamond_{[0,\Delta)}$ `ban.premium_deposited(alice)`) and proceed if Bob proceeds with the next step. For example, if Bob deposits his premium, then Alice should always go ahead and escrow her asset to exchange($\Diamond_{[0,2\Delta)}$ `apr.premium_deposited(bob)` $\rightarrow$ $\Diamond_{[0,3\Delta)}$ `apr.asset_escrowed(alice)`). Alice should

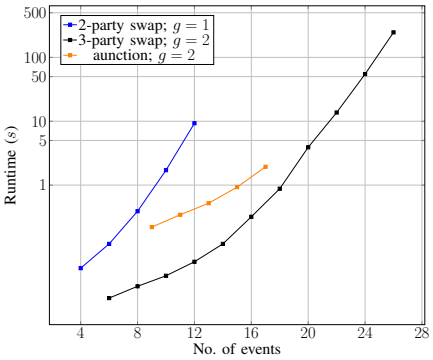


Fig. 6: Blockchain Experiments

never release her secret if she does not redeem, which means Bob should not be able to redeem unless Alice redeems, which is expressed as $\neg$`apr.asset_redeemed(bob)` $\mathcal{U}$ `ban.asset_redeemed(alice)`:

$$\begin{aligned}\varphi_{\text{alice_conform}} = \Diamond_{[0,\Delta)}\, &\texttt{ban.premium_deposited(alice)} \wedge \\ \big(\Diamond_{[0,2\Delta)}\, &\texttt{apr.premium_deposited(bob)} \rightarrow \\ \Diamond_{[0,3\Delta)}\, &\texttt{apr.asset_escrowed(alice)}\big) \wedge \\ \big(\Diamond_{[0,4\Delta)}\, &\texttt{ban.asset_escrowed(bob)} \rightarrow \\ \Diamond_{[0,5\Delta)}\, &\texttt{ban.asset_redeemed(alice)}\big) \wedge \\ \big(&\neg\texttt{apr.asset_redeemed(bob)}\ \mathcal{U} \\ &\texttt{ban.asset_redeemed(alice)}\big)\end{aligned}$$

By definition, safety means a conforming party does not end up with a negative payoff. We track the assets transferred from parties and transferred to parties in our logs. Thus, a conforming party is safe. e.g. Alice, is specified as safe φ_{alice_safety}:

$$\varphi_{\text{alice_safety}} = \varphi_{\text{alice_conform}} \rightarrow \Big(\sum_{\texttt{TransTo = alice}} \texttt{amount} \geq \sum_{\texttt{TransFrom = alice}} \texttt{amount}\Big)$$

To enable a conforming party to hedge against the sore loser attack if they escrow assets to exchange which is refunded in the end, our protocol should guarantee the aforementioned party get a premium as compensation, which is expressed as $\varphi_{\text{alice_hedged}}$:

$$\begin{aligned}\varphi_{\text{alice_hedged}} = \Diamond\big(&\varphi_{\text{alice_conform}} \wedge \\ &\texttt{apr.asset_escrowed(alice)} \wedge \\ &\texttt{apr.asset_refunded(any)}\big) \rightarrow \\ \Diamond\Big(&\sum_{\texttt{TransferTo = alice}} \texttt{amount} \geq \\ &\sum_{\texttt{TransferFrom = alice}} \texttt{amount} \\ &+ \texttt{apr.premium.amount}\Big)\end{aligned}$$

3) Analysis of Results: We put our monitor to test the traces generated by the Truffle-Ganache framework. To monitor the 2-party swap protocol we do not divide the trace into multiple segments due to the low number of events that are involved in the protocol. On the other hand, both 3-party swap and auction protocol involve a higher number of events and thus we divide the trace into two segments ($g = 2$). In Fig. 6, we show how the runtime of the monitor is effected by the number of events in each transaction log.

Additionally, we generate transaction logs with different values for deadline (Δ) and time synchronization constant (ϵ) to put the safety of the protocol in jeopardy. We observe both `true` and `false` verdict when $\epsilon \gtrapprox \Delta$. This is due to the non deterministic time stamp owning to the assumption of a partially synchronous system. The observed time stamp of each event can at most be off by ϵ. Thus, we recommend not to use a value of Δ that is comparable to the value of ϵ when designing the smart contract.

VII. Related Work

Centralized and decentralized online predicate detection in an *asynchronous* distributed system have been extensively studied (e.g., [27], [28]). Extensions to include temporal operators appear in [29], [30]. The line of work in [27]–[31] considers a fully asynchronous system. An SMT-based predicate detection solution has been introduced in [32]. On the other hand, runtime monitoring for *synchronous* distributed system has been studied in [33]–[35]. This approach has shortcoming, the major one being the assumption of a common global clock shared among all the processes. Finally, fault-tolerant monitoring, where monitors can crash, has been investigated in [36] for asynchronous and in [37] for synchronized distributed processes.

Runtime monitoring of time sensitive distributed system has been studied in [14], [15], [38], [39] and security vulnerabilities posed by blockchains have also been extensively studied in [40]–[44]. However, these methods fall short to verify the correctness of cross-chain protocols, due to their assumption regarding synchronous systems and the presence of a global clock. On the contrary, we assume the presence of a clock synchronization algorithm which limits the maximum clock skew among processes (blockchain in this context) to a constant. This is a realistic assumption since different blockchains have their own local clock and it is certain to have a skew between them. A similar SMT-based solution was studied for LTL specifications in [16], which we extend to include a more expressive time bounded logic relevant to the usage we mention in this paper.

VIII. Conclusion

In this paper, we study distributed runtime verification. We propose a technique which takes an MTL formula and a distributed computation as input. By assuming partial synchrony among all processes, first we chop the computation into several segments and then apply a progression-based formula rewriting monitoring algorithm implemented as an SMT decision problem in order to verify the correctness of the distributed system with respect to the formula. We conducted extensive synthetic experiments on traces generated by the tool UPPAAL and a set of blockchain smart contracts.

For future work, we plan to study the trade off between accuracy and scalability of our approach. Another important extension of our work is distributed runtime verification where the processes are dynamic, i.e., the process can crash and can also restore its state at any given time during execution. This will let us study a wide range of applications including airspace monitoring.

REFERENCES

[1] Y. Lu, "The blockchain: State-of-the-art and research challenges," *Journal of Industrial Information Integration*, vol. 15, pp. 80–90, 2019.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[3] M. Herlihy, "Atomic cross-chain swaps," in *Proceedings of the 2018 ACM symposium on principles of distributed computing*, 2018, pp. 245–254.

[4] M. Herlihy, B. Liskov, and L. Shrira, "Cross-chain deals and adversarial commerce," *The VLDB Journal*, pp. 1–19, 2021.

[5] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *Ieee Access*, vol. 4, pp. 2292–2303, 2016.

[6] J. Xu, K. Xue, S. Li, H. Tian, J. Hong, P. Hong, and N. Yu, "Healthchain: A blockchain-based privacy preserving scheme for large-scale health data," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8770–8781, 2019.

[7] L. W. Cong and Z. He, "Blockchain disruption and smart contracts," *The Review of Financial Studies*, vol. 32, no. 5, pp. 1754–1797, 2019.

[8] C. Dannen, *Introducing Ethereum and solidity*. Springer, 2017, vol. 318.

[9] J. Ellul and G. J. Pace, "Runtime verification of ethereum smart contracts," in *2018 14th European Dependable Computing Conference (EDCC)*. IEEE, 2018, pp. 158–163.

[10] P. Technologies, As of 2017. [Online]. Available: https://github.com/paritytech/parity

[11] R. Koymans, "Specifying Real-Time Properties with Metric Temporal Logic," *RealTime Systems*, vol. 2, no. 4, pp. 255–299, 1990.

[12] Y. Xue and M. Herlihy, "Hedging against sore loser attacks in cross-chain transactions," *arXiv preprint arXiv:2105.06322*, 2021.

[13] D. Mills, "Network time protocol version 4: Protocol and algorithms specification," Internet Requests for Comments, RFC Editor, RFC 5905, June 2010.

[14] D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu, "Monitoring metric first-order temporal properties," *J. ACM*, vol. 62, no. 2, may 2015. [Online]. Available: https://doi.org/10.1145/2699444

[15] J. Worrell, J. Ouaknine, and H.-M. Ho, "On the expressiveness and monitoring of metric temporal logic," *Logical Methods in Computer Science*, vol. 15, 2019.

[16] R. Ganguly, A. Momtaz, and B. Bonakdarpour, "Distributed Runtime Verification Under Partial Synchrony," in *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*, vol. 184, 2021, pp. 20:1–20:17. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/13505

[17] A. Momtaz, N. Basnet, H. Abbas, and B. Bonakdarpour, "Predicate monitoring in distributed cyber-physical systems," in *Proceedings of the 21st International Conference on Runtime Verification (RV)*, 2021, pp. 3–22.

[18] K. G. Larsen, P.Pattersson, and W. Yi, "UPPAAL in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[19] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, p. 558–565, jul 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[20] R. Alur and T. A. Henzinger, "Logics and models of real time: A survey," in *Real-Time: Theory in Practice*, J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 74–106.

[21] ——, "A really temporal logic," *J. ACM*, vol. 41, no. 1, p. 181–203, jan 1994. [Online]. Available: https://doi.org/10.1145/174644.174651

[22] A. Bauer and Y. Falcone, "Decentralised ltl monitoring," in *FM 2012: Formal Methods*, D. Giannakopoulou and D. Méry, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 85–100.

[23] K. Havelund and G. Rosu, "Monitoring programs using rewriting," in *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ser. ASE '01. USA: IEEE Computer Society, 2001, p. 135.

[24] V. K. Garg, *Elements of Distributed Computing*. USA: John Wiley & Sons, Inc., 2002.

[25] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, no. 3185, 2004, pp. 200–236.

[26] T. Nolan, "Alt chains and atomic transfers," https://bitcointalk.org/index.php?topic=193281.0, May, 2013, bitcoin Forum.

[27] H. Chauhan, V. K. Garg, A. Natarajan, and N. Mittal, "A distributed abstraction algorithm for online predicate detection," in *Proceedings of the 32nd IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2013, pp. 101–110.

[28] N. Mittal and V. K. Garg, "Techniques and applications of computation slicing," *Distributed Computing*, vol. 17, no. 3, pp. 251–277, 2005.

[29] V. A. Ogale and V. K. Garg, "Detecting temporal logic predicates on distributed computations," in *Proceedings of the 21st International Symposium on Distributed Computing (DISC)*, 2007, pp. 420–434.

[30] M. Mostafa and B. Bonakdarpour, "Decentralized runtime verification of LTL specifications in distributed systems," in *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015, pp. 494–503.

[31] K. Sen, A. Vardhan, G. Agha, and G.Rosu, "Efficient decentralized monitoring of safety in distributed systems," in *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, 2004, pp. 418–427.

[32] V. T. Valapil, S. Yingchareonthawornchai, S. S. Kulkarni, E. Torng, and M. Demirbas, "Monitoring partially synchronous distributed systems using SMT solvers," in *Proceedings of the 17th International Conference on Runtime Verification (RV)*, 2017, pp. 277–293.

[33] L. M. Danielsson and C. Sánchez, "Decentralized stream runtime verification," in *Proceedings of the 19th International Conference on Runtime Verification (RV)*, 2019, pp. 185–201.

[34] C. Colombo and Y. Falcone, "Organising LTL monitors over distributed systems with a global clock," *Formal Methods in System Design*, vol. 49, no. 1-2, pp. 109–158, 2016.

[35] B. Bonakdarpour and B. Finkbeiner, "Runtime verification for hyperltl," in *Proceedings of the 16th International Conference on Runtime Verification*, 2016, pp. 41–45.

[36] B. Bonakdarpour, P. Fraigniaud, S. Rajsbaum, D. A. Rosenblueth, and C. Travers, "Decentralized asynchronous crash-resilient runtime verification," in *Proceedings of the 27th International Conference on Concurrency Theory (CONCUR)*, 2016, pp. 16:1–16:15.

[37] L. Lamport and N. Lynch, *Handbook of Theoretical Computer Science*. Amsterdam: Elsevier Science Publishers B. V., 1990, vol. B, ch. 18: Distributed Computing: Models and Methods.

[38] D. Basin, F. Klaedtke, and S. Müller, "Monitoring security policies with metric first-order temporal logic," in *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 23–34. [Online]. Available: https://doi.org/10.1145/1809842.1809849

[39] P. Thati and G. Roşu, "Monitoring algorithms for metric temporal logic specifications," *Electron. Notes Theor. Comput. Sci.*, vol. 113, no. C, p. 145–162, jan 2005.

[40] A. García, E. Cambronero, C. Colombo, L. Díaz, and G. Pace, *Runtime Verification of Contracts with Themulus*, 09 2020, pp. 231–246.

[41] S. Azzopardi, J. Ellul, and G. J. Pace, "Runtime monitoring processes across blockchains," in *Fundamentals of Software Engineering*, H. Hojjat and M. Massink, Eds. Cham: Springer International Publishing, 2021, pp. 142–156.

[42] S. Azzopardi, G. Pace, F. Schapachnik, and G. Schneider, "On the specification and monitoring of timed normative systems," in *Runtime Verification*, L. Feng and D. Fisman, Eds. Cham: Springer International Publishing, 2021, pp. 81–99.

[43] X. Chen, D. Park, and G. Roşu, "A language-independent approach to smart contract verification," in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, T. Margaria and B. Steffen, Eds. Cham: Springer International Publishing, 2018, pp. 405–413.

[44] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu, "A formal verification tool for ethereum vm bytecode," in *Proceedings*

of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 912–915. [Online]. Available: https://doi.org/10.1145/3236024.3264591

IX. Appendix

Here, in Section IX-A we explain how the different UPPAAL models work and in Section IX-B we dive into the MTL specifications we use to verify 3-party swap and the auction protocol.

A. UPPAAL Models

Below we explain in details how each of the UPPAAL models work. In respect to our monitoring algorithm, we consider multiple instances of each of the models as different processes. Each event consists of the action that was taken along with the time of occurrence of the event. In addition to this, we assume a unique clock for each instance, synchronized by the presence of a clock synchronization algorithm with a maximum clock skew of ϵ.

a) The Train-Gate: It models a railway control system which controls access to a bridge for several trains. The bridge can be considered as a shared resource and can be accessed by one train at a time. Each train is identified by a unique `id` and whenever a new train appears in the system, it sends a `appr` message along with it's `id`. The Gate controller has two options: (1) send a `stop` message and keep the train in waiting state or (2) let the train cross the bridge. Once the train crosses the bridge, it sends a `leave` message signifying the bridge is free for any other train waiting to cross.

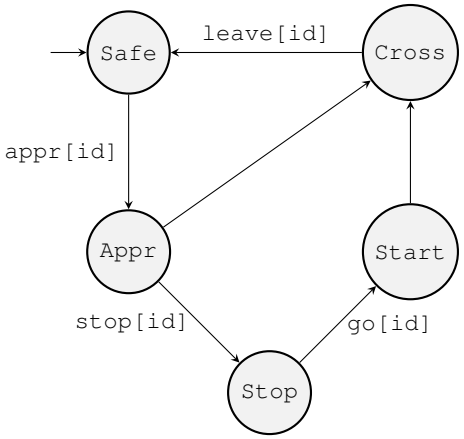


Fig. 7: Train model

The gate keeps track of the state of the bridge, in other words the gate acts as the controller of the bridge for the trains. If the bridge is currently not being used, the gate immediately offers any train appearing to go ahead, otherwise it sends a `stop` message. Once the gate is free again from a train leaving the bridge, it sends out a `go` message to any train that had appeared in the mean time and was waiting in the queue.

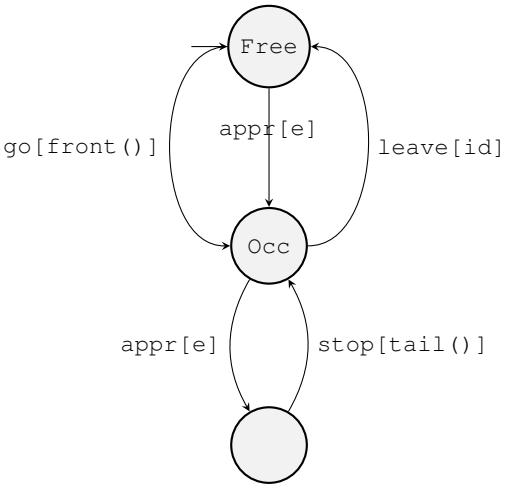


Fig. 8: Gate model

b) The Fischer's Protocol: It is a mutual exclusion protocol designed for n processes. A process always sends in a request to enter the critical section (`cs`). On receiving the request, a unique `pid` is generated and the process moves to a `wait` state. A process can only enter into the critical section when it has the correct `id`. Upon exiting the critical section, the process resets the `id` which enables other processes to enter the `cs`

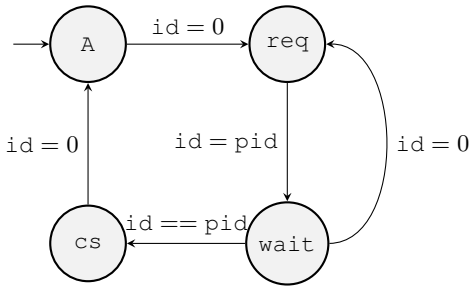


Fig. 9: Fischer model

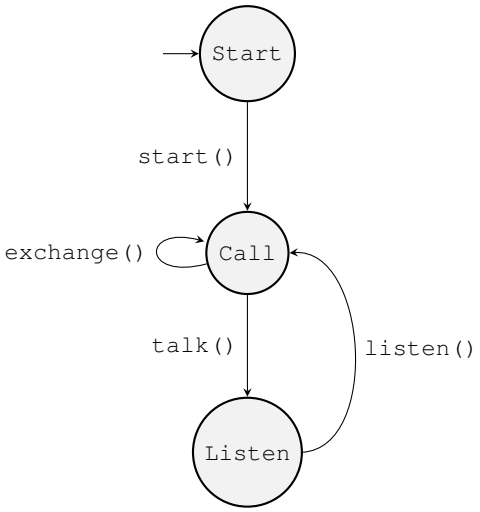


Fig. 10: Gossiping people model

c) The Gossiping People: The model consists of n people, each having a private secret they wish to share with each other. Each person can `Call` another person and after a conversation, both person mutually knows about all their secrets. With respect to our monitoring problem, we make sure that each person generates a new secret that needs to be shared among others infinitely often.

B. Blockchain

Below shows the specifications we used to verify the correctness of hedged three-party swap and auction protocols, as shown in [12]. The structure of the specifications are similar to that of hedged two-party swap protocol.

1) Hedged 3-Party Swap Protocol: The three-party swap example we implemented can be described as a digraph where there are directed edges between Alice, Bob and Carol. For simplicity, we consider each party transfers 100 assets. Transfer between Alice and Bob is called $ApricotSwap$, meaning Alice proposes to transfer 100 apricot tokens to Bob, transfer between Bob and Carol called $BananaSwap$, meaning Bob proposes to transfer 100 banana tokens to Carol, transfer between Carol and Alice, called $CherrySwap$, meaning Carol proposes to transfer 100 cherry tokens to Alice. Different tokens are managed by different blockchains (Apricot, Banana and Cherry respectively).

We denote the time they reach an agreement of the swap as $startTime$. Δ is the maximum time for parties to observe the state change of contracts by others and take a step to make changes on contracts. According of the protocol, the execution should follow the following steps:

- Step 1. Alice deposits 3 tokens as $escrow_premium$ in $ApricotSwap$ before Δ elapses after $startTime$.
- Step 2. Bob deposits 3 tokens as $escrow_premium$ in $BananaSwap$ before 2Δ elapses after $startTime$.
- Step 3. Carol deposits 3 tokens as $escrow_premium$ in $CherrySwap$ before 3Δ elapses after $startTime$.
- Step 4. Alice deposits 3 tokens as $redemption_premium$ in $CherrySwap$ before 4Δ elapses after $startTime$.
- Step 5. Carol deposits 2 tokens as $redemption_premium$ in $BananaSwap$ before 5Δ elapses after $startTime$.
- Step 6. Bob deposits 1 token as $redemption_premium$ in $ApricotSwap$ before 6Δ elapses after $startTime$.
- Step 7. Alice escrows 100 ERC20 tokens to $ApricotSwap$ before 7Δ elapses after $startTime$.
- Step 8. Bob escrows 100 ERC20 tokens to $BananaSwap$ before 8Δ elapses after $startTime$.
- Step 9. Carol escrows 100 ERC20 tokens to $CherrySwap$ before 9Δ elapses after $startTime$.
- Step 10. Alice sends the preimage of the hashlock to $CherrySwap$ to redeem Carol's 100 tokens before 10Δ elapses after $startTime$.
- Step 11. Carol sends the preimage of the hashlock to $BananaSwap$ to redeem Bob's 100 tokens before 11Δ elapses after $startTime$.
- Step 12. Bob sends the preimage of the hashlock to $ApricotSwap$ to redeem Alice's 100 tokens before 12Δ elapses after $startTime$.

If all parties are conforming, the protocol is executed as above. Otherwise, some asset refund and premium redeem events will be triggered to resolve the case where some party deviates. To avoid distraction, we do not provide details here.

a) Liveness: Below shows the specification to liveness, if all the steps of the protocol has been taken:

$$
\begin{aligned}
\varphi_{liveness} = {} & \Diamond_{[0,\Delta)}\,\texttt{apr.depositEscrowPr(alice)} \\
& \wedge \Diamond_{[0,2\Delta)}\,\texttt{ban.depositEscrowPr(bob)} \\
& \wedge \Diamond_{[0,3\Delta)}\,\texttt{che.depositEscrowPr(carol)} \\
& \wedge \Diamond_{[0,4\Delta)}\,\texttt{che.depositRedemptionPr(alice)} \\
& \wedge \Diamond_{[0,5\Delta)}\,\texttt{ban.depositRedemptionPr(carol)} \\
& \wedge \Diamond_{[0,6\Delta)}\,\texttt{apr.depositRedemptionPr(bob)} \\
& \wedge \Diamond_{[0,7\Delta)}\,\texttt{apr.assetEscrowed(alice)} \\
& \wedge \Diamond_{[0,8\Delta)}\,\texttt{ban.assetEscrowed(bob)} \\
& \wedge \Diamond_{[0,9\Delta)}\,\texttt{che.assetEscrowed(carol)} \\
& \wedge \Diamond_{[0,10\Delta)}\,\texttt{che.hashlockUnlocked(alice)} \\
& \wedge \Diamond_{[0,11\Delta)}\,\texttt{ban.hashlockUnlocked(carol)} \\
& \wedge \Diamond_{[0,12\Delta)}\,\texttt{apr.hashlockUnlocked(bob)} \\
& \wedge \Diamond\,\texttt{assetRedeemed(alice)} \\
& \wedge \Diamond\,\texttt{assetRedeemed(bob)} \\
& \wedge \Diamond\,\texttt{assetRedeemed(carol)} \\
& \wedge \Diamond\,\texttt{EscrowPremiumRefunded(alice)} \\
& \wedge \Diamond\,\texttt{EscrowPremiumRefunded(bob)} \\
& \wedge \Diamond\,\texttt{EscrowPremiumRefunded(carol)} \\
& \wedge \Diamond\,\texttt{RedemptionPremiumRefunded(alice)} \\
& \wedge \Diamond\,\texttt{RedemptionPremiumRefunded(bob)} \\
& \wedge \Diamond\,\texttt{RedemptionPremiumRefunded(carol)}
\end{aligned}
$$

b) Safety: Below shows the specification to check if an individual party is conforming. If a party is found to be conforming we ensure that there is no negative payoff for the corresponding party. Specification to check Alice is

conforming:

$$\begin{aligned}
\varphi_{alice_conf} = &\Diamond_{[0,\Delta)}\,\texttt{apr.depositEscrowPr(alice)}\\
&\wedge\big(\Diamond_{[0,3\Delta)}\,\texttt{che.depositEscrowPr(carol)}\rightarrow\\
&\Diamond_{[0,4\Delta)}\,\texttt{che.depositRedemptionPr(alice)}\big)\\
&\wedge\big(\neg\texttt{che.depositRedemptionPr(alice)}\,\mathcal{U}\\
&\texttt{che.depositEscrowPr(carol)}\big)\wedge\\
&\big(\Diamond_{[0,6\Delta)}\,\texttt{apr.depositRedemptionPr(bob)}\rightarrow\\
&\Diamond_{[0,7\Delta)}\,\texttt{apr.assetEscrowed(alice)}\big)\\
&\wedge\big(\neg\texttt{apr.assetEscrowed(alice)}\,\mathcal{U}\\
&\texttt{apr.depositRedemptionPr(bob)}\big)\\
&\wedge\big(\Diamond_{[0,9\Delta)}\,\texttt{che.assetEscrowed(carol)}\rightarrow\\
&\Diamond_{[0,10\Delta)}\,\texttt{che.hashlockUnlocked(alice)}\big)\\
&\wedge\big(\neg\texttt{che.hashlockUnlocked(alice)}\,\mathcal{U}\\
&\texttt{che.assetEscrowed(carol)}\big)\wedge\\
&\big(\neg\texttt{ban.hashlockUnlocked(carol)}\,\mathcal{U}\\
&\texttt{che.hashlockUnlocked(alice)}\big)\\
&\wedge\big(\neg\texttt{apr.hashlockUnlocked(bob)}\,\mathcal{U}\\
&\texttt{che.hashlockUnlocked(alice)}\big)
\end{aligned}$$

Specification to check conforming Alice does not have a negative payoff:

$$\varphi_{alice_safety} = \varphi_{alice_conform}\rightarrow\Big(\sum_{\texttt{TransTo = alice}}\texttt{amount}\geq\sum_{\texttt{TransFrom = alice}}\texttt{amount}\Big)$$

c) Hedged: Below shows the specification to check that, if a party is conforming and its escrowed asset is refunded, then it gets a premium as compensation.

$$\begin{aligned}
\varphi_{alice_hedged} = &\Diamond\big(\varphi_{alice_conform}\\
&\wedge\texttt{apr.assetEscrowed(alice)}\big)\\
&\rightarrow\Diamond\Big(\sum_{\texttt{TransTo = alice}}\texttt{amount}\\
&\geq\sum_{\texttt{TransFrom = alice}}\texttt{amount}\\
&+\texttt{apr.redemptionPremium.amount}\Big)
\end{aligned}$$

2) Auction Protocol: In the auction example, we consider Alice to be the auctioneer who would like to sell a ticket (worth 100 ERC20 tokens) on the ticket (`tckt`) blockchain, and Bob and Carol bid on the `coin` blockchain and the winner should get the ticket and pay for the auctioneer what they bid, and the loser will get refunded. We denote the time that they reach an agreement of the auction as $startTime$. Δ is the maximum time for parties to observe the state change of contracts by others and take a step to make changes on contracts. Let $TicketAuction$ be a contract managing the "ticket" on the ticket blockchain, and $CoinAuction$ be a contract managing the bids on the coin blockchain. The protocol is briefed as follows.

- Setup. Alice generates two hashes $h(s_b)$ and $h(s_c)$. $h(s_b)$ is assigned to Bob and $h(s_c)$ is assigned to Carol. If Bob is the winner, then Alice releases s_b. If Carol is the winner, then Alice releases s_c. If both s_b and s_c are released in $TicketAuction$, then the ticket is refunded. If both s_b and s_c are released in $CoinAuction$, then all coins are refunded. In addition, Alice escrows her ticket as 100 ERC20 tokens in $TicketAuction$ and deposits 2 tokens as premiums in $CoinAuction$.
- Step 1 (Bidding). Bob and Carol bids before Δ elapses after $startTime$.
- Step 2 (Declaration). Alice sends the winner's secret to both chains to declare a winner before 2Δ elapses after $startTime$.
- Step 3 (Challenge). Bob and Carol challenges if they see two secrets or one secret missing, i.e. Alice cheats, before 4Δ elapses after $startTime$. They challenge by forwarding the secret released by Alice using a path signature scheme [3].
- Step 4 (Settle). After 4Δ elapses after $startTime$, on the $CoinAuction$, if only the hashlock corresponding to the actual winner is unlocked, then the winner's bid goes to Alice. Otherwise, the winner's bid is refunded. Loser's bid is always refunded. If the winner's bid is refunded, all bidders including the loser gets 1 token as premium to compensate them. On the $TicketAuction$, if only one secret is released, then the ticket is transferred to the corresponding party who is assigned the hash of the secret. Otherwise, the ticket is refunded.

a) Liveness: Below shows the specification to check that, if all parties are conforming, the winner (Bob) gets the ticket and the auctioneer gets the winner's bid.

$$\begin{aligned}
\varphi_{liveness} = &\Diamond_{[0,\Delta)}\,\texttt{coin.bid(bob)}\\
&\wedge\Diamond_{[0,2\Delta)}\,\texttt{coin.declaration(alice, }s_b\texttt{)}\\
&\wedge\Diamond_{[0,2\Delta)}\,\texttt{tckt.declaration(alice, }s_b\texttt{)}\\
&\wedge\Diamond_{(4\Delta,\infty)}\,\texttt{coin.redeemBid(any)}\\
&\wedge\Diamond_{(4\Delta,\infty)}\,\texttt{coin.refundPremium(any)}\\
&\wedge\big(\texttt{coin.bid(carol)}\rightarrow\\
&\Diamond_{[0,\Delta)}\,\texttt{coin.refundBid(any)}\big)\\
&\wedge\texttt{tckt.redeemTicket(any)}\\
&\wedge\neg\texttt{coin.challenge(any)}\\
&\wedge\neg\texttt{tckt.challenge(any)}
\end{aligned}$$

b) Safety: Below shows the specification to check that, if a party is conforming, this party does not end up worse off. Take Bob (the winner) for example.

Specification to define Bob is conforming:

$$
\begin{aligned}
\varphi_{bob_conform} = \Diamond_{[0,\Delta)}\ & \texttt{coin.bid(bob)} \\
& \wedge \Big(\big(\texttt{coin.declaration(alice, } s_c\texttt{)} \vee \\
& \texttt{coin.challenge(carol, } s_c\texttt{)} \big) \rightarrow \\
& \wedge \big(\texttt{tckt.declaration(alice, } s_c\texttt{)} \vee \\
& \texttt{tckt.challenge(carol, } s_c\texttt{)} \vee \\
& \texttt{tckt.challenge(bob, } s_c\texttt{)} \big) \Big) \\
& \wedge \Big(\big(\texttt{coin.declaration(alice, } s_b\texttt{)} \vee \\
& \texttt{coin.challenge(carol, } s_b\texttt{)} \big) \rightarrow \\
& \wedge \big(\texttt{tckt.declaration(alice, } s_b\texttt{)} \vee \\
& \texttt{tckt.challenge(carol, } s_b\texttt{)} \vee \\
& \texttt{tckt.challenge(bob, } s_b\texttt{)} \big) \Big) \\
& \wedge \Big(\big(\texttt{tckt.declaration(alice, } s_c\texttt{)} \vee \\
& \texttt{tckt.challenge(carol, } s_c\texttt{)} \big) \rightarrow \\
& \wedge \big(\texttt{coin.declaration(alice, } s_c\texttt{)} \vee \\
& \texttt{coin.challenge(carol, } s_c\texttt{)} \vee \\
& \texttt{coin.challenge(bob, } s_c\texttt{)} \big) \Big) \\
& \wedge \Big(\big(\texttt{tckt.declaration(alice, } s_b\texttt{)} \vee \\
& \texttt{tckt.challenge(carol, } s_b\texttt{)} \big) \rightarrow \\
& \wedge \big(\texttt{coin.declaration(alice, } s_b\texttt{)} \vee \\
& \texttt{coin.challenge(carol, } s_b\texttt{)} \vee \\
& \texttt{coin.challenge(bob, } s_b\texttt{)} \big) \Big)
\end{aligned}
$$

Specification to define Bob does not end up worse off:

$$
\begin{aligned}
\varphi_{bob_safety} = \varphi_{bob_conform} \rightarrow & \\
& \Diamond \Big(\big(\texttt{coin.refundBid(any)} \\
& \wedge \texttt{coin.redeemPremium(any)} \big) \vee \\
& \texttt{tckt.redeemTicket(any)} \Big)
\end{aligned}
$$

c) Hedged: Below shows the specification to check that, if a party is conforming and its escrowed asset is refunded, then it gets a premium as compensation.

$$
\begin{aligned}
\varphi_{bob_hedged} = \Box \Big(& \varphi_{bob_conforming} \\
& \wedge \big(\texttt{tckt.refundTicket(alice)} \vee \\
& \texttt{tckt.redeemTicket(carol)} \big) \Big) \rightarrow \\
& \Diamond \big(\texttt{coin.refundBid(any)} \\
& \wedge \texttt{coin.redeemPremium(any)} \big)
\end{aligned}
$$