Efficient On-line Detection of Temporal Patterns

Shlomi Dolev\(^1\), Jonathan Goldfeld\(^1\), and Rami Puzis\(^2\)

\(^1\) Ben-Gurion University, Department of Computer Science, Beer-Sheva, Israel
\(^2\) Deutsche Telekom Laboratories, Ben-Gurion University, Beer-Sheva, Israel

Abstract. Identifying a temporal pattern of events is a fundamental task of on-line (real-time) verification. We present efficient schemes for on-line monitoring of events for identifying desired/undesired patterns of events. The schemes use preprocessing to ensure that the number of comparisons during run-time is minimized. In particular, the first comparison following the time point when an execution sub-sequence cannot be further extended to satisfy the temporal requirements, halts the process that monitors the sub-sequence.

1 Introduction

Many complex systems, both hardware and software, require a sound verification of their operation, usually in the form of safety and liveness properties. One of the prominent formal verification methods used today is model-checking, which models the system as a state-transition system, and performs an exhaustive search of its state-transition graph for possible runs where desired properties do not hold. One of the drawbacks of model checking is the state explosion problem. Explosion due to the need to explore an exponential number of states with relation to the number of system variables. This yields a costly check. Another bold problem is in the modeling process itself. The verification is only as good as the model of the (actual hardware and software) implementation rather than the implementation itself.

An even older classical method to verify the system specifications is testing. While testing examines the implemented system (rather than only a model of the system), testing cannot be exhaustive since the number of possible input sequences grows exponentially with the sequence length. Of course an exhaustive search for specifications’ violation is not feasible. Thus, run time verification may assist in coping with implementation flaws that materialize only during (specific rare) executions.

We concentrate on a specific task of specifications’ run-time verification, namely the detection of temporal patterns. Rather than modeling the system and searching its entire state-transition graph, as done in model checking, we devise an approach toward verification of time restrictions over sequences of events at system run-time. We use temporal constraints semantics to describe the specification.

Our Contribution. We design an algorithm which, given a description of the system as a set of possible events and a specification of safety properties as un-
desired temporal patterns, monitors the system and detects when such patterns occur.

First, we describe how to preprocess the input patterns in graph form to get their minimized representation by removing redundant constraints. Then we describe a many-state automaton which tracks pattern prefixes using tokens and detects pattern completion. This automaton tracks the system events efficiently. Once a token does not adhere to some temporal constraint, the token is discarded within one comparison. In addition an earliest possible notification on a completed pattern is given.

Related Work. Run time verification of asynchronous systems for ensuring the safety and liveness specification by monitoring events and states in run-time were discussed in e.g., [9,3,4]. In contrast we are interested in monitoring timed events in synchronous (or semi-synchronous real-time) systems. The preprocessing phase resembles results in Temporal Constraint Networks [8]. In contrast, [8] does not consider the on-line monitoring task and its composition with the preprocessing phase. In order to detect desired/undesired patterns of events we employ methods similar to the classical string matching solutions [6]. In contrast to matching simple strings or subsequences where only the order of elements is important, in our case system events must occur in a timely manner to fit a given pattern.

A fundamental discussion of model checking appears in [7], and a popular method to address the state explosion problem of model checking is described in [2], by representing the state transition graph using propositional logic formulae. an approach of extending CTL to include timing constraints appears in [1]. In [10] transitional durations are added to timed temporal formulae as an extension of Kripke Structures, and timed versions of CTL are considered. These frameworks however, do not address real-time verification.

Organization. The definitions of system events, executions, and temporal patterns appear in the next section, as part of the study of continuous temporal patterns. In Sect. [3] we study the less restricted definition of sporadic temporal patterns by analyzing the pattern’s temporal constraints semantics and devising a detection algorithm. Finally, concluding remarks appear in in Sect. [4]

2 Continuous Temporal Patterns

An event \( a \) is a discrete input to our system taken from a given set of possible inputs, called the event type set. A timed event, \( te \), is a tuple \((a, t)\) such that \( a \) is an event type and \( t \) is the time at which an event of type \( a \) took place. Similarly to multi-event segments in simulation theory [13], we define an execution of a system as an input stream of timed events \( te_1 = (a_1, t_1), te_2 = (a_2, t_2), ... \), such that for every two timed events \( te_i \) and \( te_{i+1} \) it holds that \( t_i < t_{i+1} \). We assume that events in the system occur in discrete time points.

A temporal pattern \( TP \) is a tuple \((A, C)\), \( A = (a_1, a_2, \ldots, a_n) \) is a sequence of typed (non-timed) events. Each \( a_i \) has a type \( a_i.type \) from the event type set. \( C \) is a set of temporal constraints, such that each \( c \in C \) is a tuple \((a_i, a_j, w)\),
which means that the time interval between the events \( a_i \) and \( a_j \) is at most \( w \).
We call such a constraint a max constraint and later add min constraints and handle both types of constraints. Event types are not necessarily distinct.

**Detection of Continuous Temporal Patterns.** We track a stream of timed events to identify temporal patterns that respect the constraints of \( TP \). In this section we consider the simple case of a continuous pattern. For every \( 1 < i < n - 1 \) we call \( a_i \) and \( a_{i+1} \) consecutive events. A continuous pattern match is an execution sub-sequence where consecutive events happen successively without intervening events. If the temporal pattern events are \( a_1, \ldots, a_n \), then we wish to detect when this pattern (along with the temporal constraints) occurs with no event in between. A (type-wise\(^3\)) execution \( a_1, \ldots, a_i, b, a_{i+1}, \ldots, a_n \) where \( b.type \neq a_{i+1}.type \) does not count.

**Directed Graphs and Tokens.** A temporal pattern \((A, C)\) is represented by a pattern graph. The events in \( A \) define the graph nodes, and the constraints in \( C \) define edges. A constraint \((a_i, a_j, w)\) is a directed edge from \( a_i \) to \( a_j \) with weight \( w \). If there is no constraint between \( a_i \) and \( a_{i+1} \) for some \( i \), we define an edge \((a_i, a_{i+1}, \infty)\), where \( \infty \) signifies any finite time may pass between \( a_i \) and \( a_{i+1} \). An edge between consecutive events is a simple edge, and any other edge is an overpassing edge. A path using only simple edges is a simple path. We denote \( a_i \) and \( a_j \) as \( e.src \) and \( e.dst \), respectively. \((a_1, a_2, \ldots, a_n)\) is the defined chronological order of the pattern’s events. In particular, \( G \) is a WDAG (Weighted DAG).

A preprocessing phase of pattern constraints is explained in Sect. 3.

Execution sub-sequences that partially match a pattern are represented by tokens. A token resides on an event, and carries with it a history of the time points at which it reached previous events. HandleEvent (Alg. 1) handles an event log received from the system in the form of a timed event \((type, t)\), where \( type \) is the event type and \( t \) is the time point at which it occurred. If \( tkn \) is a token currently residing on the graph, then \( tkn.evnt \) and \( tkn.h \) are the event the token is on and the token’s history, respectively. The method \( getTime(a) \) of \( tkn.h \) returns the time point at which \( tkn \) reached \( a \).

```plaintext
1 tkns := set of graph tokens;
2 ForEach tkn in tkns Do
3     b := successor of tkn.evnt;
4     If (b.type) != type Then discard tkn;
5     ForEach graph edge (a,b) Do
6         If (t - tkn.h.getTime(a) > w(a,b)) Then discard tkn;
7         add (b,t) to tkn.h;
8         tkn.evnt := b;
9     If (tkn.evnt is the last event) Then
10        report tkn;
11        discard tkn;
12        If (type matches the first event) Then
13           Add a new token;

Alg. 1. HandleEvent(type, t)
```

In (line 1) we get the set of graph tokens. We handle each such token (line 2). In (line 3) we obtain the successive event of \( tkn.evnt \). If the event’s type does not match \( type \) (the logged event’s type), we discard \( tkn \) (line 4). We check if all edge constraints whose head is \( b \) are satisfied, and if not we discard \( tkn \) (lines 5-7). If the constraints are satisfied, we update the token’s history and move \( tkn \) to the next event (lines 8-9). If \( tkn \) reached the last pattern event we report the pattern match and provide its history, and then discard \( tkn \) (lines 10-12). Finally, if \( type \)

\(^3\) \( a_j \) denotes an event of type \( a_j.type \)
matches the type of the first pattern event, we create a new token on the first event (lines 13-14).

An illustration of token tracking is shown in Fig. 1.

Fig. 1. An example of token tracking for the event input stream: 
\((a, 0), (b, 12), (a, 30), (c, 37)\). \(z_2\) becomes obsolete when \(c\) occurs. \(z_1\) reaches the last event \(c\) and completes a continuous pattern match.

3 Sporadic Temporal Patterns

3.1 Maximum Constraints

A sporadic pattern match is an execution sub-sequence where non-sequential system events may occur between consecutive pattern events. If the temporal pattern events are \(a_1, \ldots, a_n\), a (type-wise) execution \(a_1, \ldots, a_i, b, a_{i+1}, \ldots, a_n\) where \(b\) type \(\neq a_{i+1}\) type matches the pattern, as long as the temporal constraints are satisfied. We design an algorithm to detect sporadic pattern matches, so that an obsolete token is detected as soon as it becomes obsolete, essentially at the very next comparison of the token against some constraint, and can therefore be discarded.

We shall describe two phases of the pattern detection paradigm. A preprocessing phase and an on-line detection phase. The preprocessing phase will yield an equivalent temporal pattern in a more restrictive form. The on-line detection phase will utilize a many-state automaton to keep track during the system’s operation of execution sub-sequences partially matching a pattern. The sub-sequences are represented by tokens that reside on the automaton’s states.

Preprocessing Phase.

We reduce the pattern constraint set while maintaining the temporal constraints semantics. We describe the preprocessing process. For every \(1 \leq i < n\) we add to \(G\) an edge \((a_{i+1}, a_i, 0)\), which we simply call a 0-edge. We compute the shortest path from \(a_i\) to \(a_j\) for \(1 < i < n - 1\) and \(i + 1 < j < n\). This can be done with
$O(n^3)$ comparisons using the Floyd-Warshall algorithm. We then update $G$ by removing every overpassing edge that is not the unique shortest path between its ends. In addition, we update the weight of every simple (non overpassing) edge to be the weight of the shortest path between its ends. This update takes $O(n^2)$ comparisons. We denote the ensuing graph $G'$. An example of a graph before, during and after preprocessing can be seen in Figs. 2, 3 and 4 respectively.

![Fig. 2. A graph $G$ representing an input temporal pattern](image1)

![Fig. 3. $G$ with backward 0-edges added](image2)

![Fig. 4. The graph $G'$ which is equivalent to $G$ with regard to temporal constraints semantics](image3)

**Histories and Graph Preorder.** We call every single update of $G$ an update step. In the following discussion the graphs are temporal patterns of $n$ events, and the histories consist of these events. A *history* is a tuple $H = ((a_1, t_1), ..., (a_n, t_n))$, where $t_j$ is the time-point at which $a_j$ occurred. We denote $t_j$ as $H(a_j)$. We denote $t_j - t_i$ as $H(a_i, a_j)$ or as $H(e)$ where $e = (a_i, a_j, w)$.
for some \( w \). A history \( H \) fits a graph \( G \) if: for every \( (a_i, t_i) \) and \( (a_j, t_j) \) in \( H \), if \( (a_i, a_j, t) \) is a weighted edge in \( G \), then \( H(a_i, a_j) \leq t \). Alternately, we say \( H \) fits the underlying temporal pattern \( TP \).

We define a preorder \( \preceq \) (a reflexive and transitive relation) between graphs as follows: \( G_1 \preceq G_2 \) if for every history \( H \): \( (H \text{ fits } G_1) \) implies that \( (H \text{ fits } G_2) \). It is easy to show that \( \preceq \) is indeed a preorder. \( \preceq \) naturally induces an equivalence relation \( \equiv \) on graphs: \( G_1 \equiv G_2 \) if \( G_1 \preceq G_2 \) and \( G_2 \preceq G_1 \). In the following discussion, equivalence of graphs refers to the graphs modulo the backward 0-edges.

**Proposition 1.** Let \( G \) and \( G' \) be the graphs before and after the execution of the preprocessing process, respectively. Then the following properties hold:

- **Equivalence:** \( G \equiv G' \).
- **Minimality:** \( G' \) is the minimal graph that is equivalent to \( G \). If any overpassing edge is removed from \( G' \) or if the weight of some edge is reduced in \( G' \), then it will no longer be equivalent to \( G \).

Note that for simplicity we refer to \( G \) and \( G' \) as the respective graphs both with and without the 0-edges: with 0-edges when referring to paths, and without 0-edges when referring to history fittings and graph equivalences.

See Appendix A. for proof. This result is a special case of the results in Subsect. 3.2 and follows the result in [8].

**On-line Detection Phase.**

We build an automaton \( TA \) whose states are the pattern events. We may consider the graph as a representation of the automaton, with its events as the different states. At any given time in the execution, an event may hold several tokens. A token represents a sub-sequence of the execution, partially matching the temporal pattern. The only transitions are from an event \( a_i \) to the next event \( a_{i+1} \) for every \( 1 < i < n - 1 \), where the rule of the transition is that the token must satisfy the constraint of the simple edge \( (a_i, a_{i+1}) \). Additionally, a token must satisfy other constraints as explained in the following schemes.

**Deadline Scheme.** Every token \( tkn \) is associated with a minimum heap \( tkn.DL \) (for DeadLines) which is empty at first, and with a dynamic hash table \( tkn.DLH \). An element in the heap is a tuple \((evnt, dl, ptr)\), where \( evnt \) is an event that the token is yet to reach, and \( dl \) is the deadline for the token to reach \( evnt \). If the token does not reach \( evnt \) by the time \( dl \) it becomes obsolete. \( ptr \) is a pointer to the twin element in \( tkn.DLH \). An element in \( tkn.DLH \) is a tuple \((evnt, ptr)\). The event \( evnt \) is the key, and \( ptr \) is a pointer to the twin element \( tkn.DL \).

This duplication of data is necessary to ensure that no more than one instance of some event is in the heap. This keeps the space complexity of a token’s heap and hash table to \( O(n) \), instead of a possible \( \Theta(n^2) \) for \( \Theta(n^2) \) overpassing edges. \( tkn.evnt \) is the event \( tkn \) is on. We call this scheme the **deadline scheme**.

If \( tkn \) resides on an event \( a \) and the next chronological event in the pattern \( b \) occurs at time \( t \), we first check \( tkn \) against the simple edge \( (a, b) \). If the constraint does not hold, \( tkn \) is discarded from \( TA \). HandleEvent (Alg. 2) handles an event...
log (type, t). The method `newToken(tkn, a, t)` spawns a new token on `a` which inherits `tkn`'s history and adds to the history of the new token the element `(a, t)`. `opEdges(a)` returns the set of overpassing edges whose source is `a`.

In (line 1) we get the tokens awaiting a type event, in descending chronological order. We handle each such token (line 2). In (line 3) we check the minimum element `m` of `tkn.DL`. If `m.dl < t`, `tkn` is discarded from `TA` along with its associated data structures, and we check the next token in `tkns` (line 4). Otherwise, if `b`, the next chronological event after `tkn.evnt`, holds a token, the old token is discarded (line 6). We spawn a new token `tkn'` on `b` that inherits `tkn.DL` and `tkn.DLH` (line 7). If `m.evnt = b`, we remove `m` from `tkn'.DL` and `m.ptr` from `tkn'.DLH` (line 8), since this deadline is no longer relevant. Furthermore, every overpassing edge that is outgoing from `b` may contribute a deadline to `tkn'.DL`: For every overpassing edge `e = (b, c, w)`, we check if the key `c` appears in `tkn'.DLH` (line 11). If it appears as `(c, ptr1)`, let `ptr1 = m0 = (c, dl, ptr0)`. If `t + w(e) < dl`, we change the value of `dl` to `t + w`, and then move `m0` up the heap until the heap property holds (lines 13-15). If `dl ≤ t + w(e)`, we do nothing. If `c` does not appear in `tkn'.DLH`, we create twin elements `m0 = (c, t + w, ptr0)` and `m1 = (c, ptr1)` such that `ptr1` points to `m1`, and insert them to `tkn.DL` and `tkn.DLH`, respectively (lines 16-21).

We handle the tokens by the events they are on, from late to early. The reason for this is as follows: Say we have tokens on events `ai` and `ai+1`, and the type of events `ai+1` and `ai+2` is `type`. Now a `type` event occurs. If we handle the token on `ai` first, it will discard the token on `ai+1`, although that token may spawn a token on `ai+2`. Thus we lose a possible pattern match.

The reason we spawn a new token `tkn'` on `ai+1` rather than moving `tkn` to `ai+1`, is that `ai+1` may occur again while `tkn` is still relevant, thus spawning a new token `tkn''` with different deadlines than those of `tkn'`. This algorithm ensures we detect the first instance of the temporal pattern during an execution. If we wish to detect all instances, we cannot discard a token (as in lines 8-9), since it has a unique history, and may complete to a unique instance of the temporal pattern. If \( \max\{w(a_i, a_{i+1}) \mid 1 \leq i \leq n - 1\} = k \), then there are at most \( O(k^n) \) tokens at any point during the execution.

Note that the deadline scheme is efficient for a small number of overpassing edges. In the worst case we have an edge between all pairs of events, a total of \( \Theta(n^2) \) overpassing edges in the graph. An update of a deadline in the heap and
hash table is $O(\log(n))$. Hence, the total number of operations per token may be up to $\Theta(n^2 \cdot \log(n))$ for this scheme.

**History Scheme.** Alternately to the heap and hash table data structures, we may simply keep a history of the time points when a token reaches events, and each time a token is spawned on an event, we compare the history with temporal constraints of overpassing edges whose destinations are the current event. We call this scheme the *history scheme*. Since there are $n$ events and each time we check back against $O(n)$ constraints, this adds up to $O(n^2)$ operations per token, which is better than the deadline scheme. We note that in both cases the token holds an associated data structure of $O(n)$ space.

There is still the matter of whether we detect an obsolete token as soon as it becomes so. For this to hold, we actually need to add overpassing edges to the graph after finding all shortest paths between pairs of nodes, with an edge’s weight set to the shortest path weight. The proof that this in fact ensures an earliest detection of an obsolete token is a special case of Prop. 2.

On the opposite extreme, we may consider the case presented below in Fig. 5. Here, according to the history scheme, we add overpassing edges with weight 100 between all non-consecutive events. Thus for a token traversing the entire graph we will have $\Theta(n^2)$ checks against constraints of overpassing edges, and it will use $\Theta(n)$ space. On the other hand, if we do not add edges, but maintain a heap of deadlines as in the deadline scheme, we will have only one deadline in the heap and one check against this deadline at each event the token reaches. Thus the token will have only $\Theta(n)$ checks and will use only $\Theta(1)$ space.

We in fact used the history scheme in Alg. 1 to detect continuous pattern matches. We can also benefit from using the deadline scheme to detect such patterns, when there are few overpassing edges in the pattern graph.

What is the differential line between using one scheme and not the other? A straightforward calculation follows. Let $k$ be the number of overpassing edges in the graph. Then in the deadline scheme, a token will have $O(k \cdot \log(n)) + n$ operations. In the history scheme we may have up to $O(n^2)$ overpassing edges as in the example in Fig. 5. Thus a starting point for finding the differential line is $k = O(n^2/\log(n))$. Furthermore, depending on the exact configuration of constraints and their values, it may be that in the average case a token’s heap has constant size throughout the graph traversal, as in Fig. 5 where $k = 1$. In this case we have a better number of operations and space complexity for the deadline scheme, namely $O(n)$ and $O(1)$, respectively.

### 3.2 Adding Minimum Constraints

We now broaden our scope by allowing input patterns with both maximum and minimum constraints. We highlight the differences in definitions and notations that ensue. Here a temporal pattern input is a triplet $TP = (A, CMax, CMin)$. $A$ is the sequence of events: $A = (a_1, a_2, ..., a_n)$. $CMax$ is a set of maximum temporal constraints (max constraints) on $A$. An element of $CMax$ is $(a_i, a_j, t)$, where $t$ is the maximum time allowed between $e_i$ and $e_j$. $CMin$ is a set of
minimum temporal constraints (min constraints) on \( A \). An element of \( CMin \) is \((a_i, a_j, t)\), where \( t \) is the minimum time allowed between \( e_i \) and \( e_j \).

We note that a temporal constraint may be 0 or \( \infty \). Furthermore, one may change the model slightly by allowing only strictly positive time intervals for the min constraints, i.e. each constraint is of value at least 1. This simply amounts to adding the constraints \((a_i, a_{i+1}, 1)\) to \( CMin \) for every \( 1 < i < n - 1 \). In general, we may define in our model (or get a restriction as input) that all constraints must be in the time window \((m, M)\) for some \( m, M \in \mathbb{N} \) where \( m \leq M \). This amounts to adding the constraints \((a_i, a_{i+1}, m)\) to \( CMin \) and \((a_i, a_{i+1}, M)\) to \( CMax \) for every \( 1 < i < n - 1 \).

Preprocessing Phase.

The definition of a history fitting a temporal pattern here is similar to Subsect. 3.1, only here the history should adhere to both min and max constraints. We describe the preprocessing process. We build a weighted, directed graph \( G = (A, E) \) whose nodes are the events in \( A \), and whose edges are \( E = \{(a_i, a_j, t) \mid (a_i, a_j, t) \in CMax\} \cup \{(a_i, a_j, -t) \mid (a_i, a_j, t) \in CMin\} \). Also, there’s an edge \((a_i, a_{i+1}, \infty)\) or \((a_{i+1}, a_i, 0)\) for consecutive events without a constraint in \( CMax \) or \( CMin \), respectively. The purpose of this construction is again to utilize the Floyd-Warshall algorithm for finding all shortest paths. Here we will have a "window of opportunity” for every pair of events \( a_i \) and \( a_j \) that will indicate the exact time frame in which a token residing on \( a_i \) must reach \( a_j \) to hold by the constraints. We call this window a time window for \( a_i \) and \( a_j \). We run the Floyd-Warshall algorithm to find the shortest paths between all pairs of nodes in \( G \). Our purpose is to contract the time windows as much as possible for all pairs of events. An example graph is depicted in Fig. 6.

Time Windows. Next we define the time windows. Let \( \min(a, b) \) be the weight of the shortest \((a, b)\)-path. For all \( 1 < i < j < n: tw(a_i, a_j) = (-\min(a_j, a_i), \min(a_i, a_j)) \). We denote the set of all time windows of \( TP \) as \( TW(\mathcal{TP}) \). \( a_i \) and \( a_j \) are called the source and destination of \( tw(a_i, a_j) \) respectively, and are denoted \( \text{src}(tw) \) and \( \text{dst}(tw) \), respectively. The first index of a time window \( tw \) is called the lower bound of \( tw \), denoted as \( tw \). The second index is called the upper bound, denoted as \( tw \). A time window between consecutive events \( a_i \) and \( a_{i+1} \) is called a simple time window. if \( tw_i(a_i, b_i) \) and \( tw_j(a_j, b_j) \) are time windows
such that \((a_j \leq a_i \lor a_j = a_i) \land (b_i \leq b_j \lor b_i = b_j)\), we say that \(tw_i\) is subsumed by \(tw_j\). If \(H\) is a history and \(tw\) is a time window between events \(a\) and \(b\), then \(H(tw)\) denotes \(H(a,b)\). Formally, we define \(CMax' = \{(a_i, a_j, \min(a_i, a_j) \mid a_i, a_j \in A, a_i \leq a_j)\}\), \(CMin' = \{(a_i, a_j, -\min(a_j, a_i) \mid a_i, a_j \in A, a_i \leq a_j)\}\), and \(TP' = \langle A, CMax', CMin'\rangle\), and show that \(TP \equiv TP'\). \(TP'\) is in fact the temporal constraints closure of \(TP\). These updates keep the temporal constraints semantics (Prop. 2).

Another case to consider is whether the temporal pattern is consistent; whether there is in fact a history that fits the pattern. If for example we have the constraints: \((a, b, 10), (b, c, 7) \in CMax\), and \((a, c, 24) \in CMin\), there is a contradiction in the temporal semantics of the temporal pattern, and no history can fit the pattern. However not all contradictions may manifest in so obvious a fashion. Fortunately, this happens exactly when there is a negative cycle (a cycle with negative weight) in \(G\), something we can detect with a slight modification of the Floyd-Warshall algorithm.

**Proposition 2.** Let \(TP\) and \(TP'\) be the temporal patterns before and after the execution of the preprocessing process, respectively. Then the following properties hold:

- **Equivalence:** \(TP \equiv TP'\).
- **No-Negative:** There exists a history \(H\) that fits \(TP \iff\) there is no negative cycle in \(G\).
- **Minimality:** Assuming there is no negative cycle in \(G\), then: \(TP'\) is the minimal graph that is equivalent to \(TP\), i.e. any further contraction of any time window in \(TP'\) will result in a temporal pattern \(TP''\) that is not equivalent to \(TP'\).

See Appendix A. for proof. A different form of the proof has appeared in the framework of Temporal Constraint Networks [8], and in particular the no-
negative result has appeared in \[11,12\].

**On-line Detection Phase.**
First we observe that, unlike Alg. 2 (lines 8-9), an older token from \(a_{i+1}\) cannot be discarded when a new token is spawned, since the older token may complete to a pattern match while the new one does not, and vice versa. See Fig. 7 for an illustrating example.

**Fig. 7.** An example of a temporal pattern for min and max constraints after preprocessing. Given the history \(((a, 0), (b, 6), (b, 9))\) we have two tokens on \(b\) with different histories. If \((c, 13)\) occurs \(z_1\) will advance while \(z_2\) will not, and if \((c, 17)\) occurs \(z_2\) will advance while \(z_1\) will not.

Furthermore, here we do not have overpassing edges as in Subsect. 3.1 but rather time windows. In other words we have overpassing max and min edges between all pairs of non-consecutive events, i.e. \(\Theta(n^2)\) overpassing edges. As shown in Prop. 2, a token on an event \(a_i\) whose history satisfies the temporal constraints of \(a_i\) with \(a_1, ... a_{i-1}\), may complete to a pattern match. Hence, if we use the history scheme, keeping a history for each token, then whenever the token reaches an event we can check back against constraints with earlier events. Such a simple scheme ensures the earliest detection of an obsolete token. This amounts to keeping an \(O(n)\) space history per token, and performing \(O(n^2)\) total checks per token that traverses the entire graph. Since we have \(\Theta(n^2)\) overpassing edges, it seems at a first glance, that the deadline scheme is not efficient here, since we have \(n^2 \cdot \log(n)\) operations and \(O(n)\) space per token, as in Subsect. 3.1.

However, if we examine the temporal pattern depicted in Fig. 5 where all min constraints are 0, we see that we need save only one deadline for a token and perform \(O(n)\) checks in total, which is better than the history scheme. Defining all the time windows is unnecessary and is in fact a hindrance, since the time windows do not add information; the given graph is already the temporal constraints closure of the temporal pattern. Using the history scheme here is inefficient. Roughly speaking, as the graph gets richer with more max and min constraints, the history scheme becomes more plausible, since the preprocessing
phase will yield much more information in the form of contracted time windows.

**Interleaved Max-Min Patterns.**

Let $k$ be the maximum length of a time window; $k = \max\{tw - tw : tw \in \text{TW}(TP) \land tw \text{ is a simple time window}\}$. Since we cannot discard older tokens upon spawning a new token, the number of graph tokens may be as high as $O(kn)$. We can reduce the number of graph tokens to $O(n)$ for a special class of temporal patterns, when it is enough to detect the first instance of a pattern match. An Interleaved Max-Min Temporal Pattern (IMM temporal pattern) is a temporal pattern $TP$ such that the following hold:

- $TP$ is a concatenation of pattern intervals $\{TP_i = (A_i, C_i)\}_{i=1}^r$. For all $1 \leq i \leq r-1$ the last event in $A_i$ is the first event in $A_{i+1}$.
- $C = (\bigcup_{i=1}^r C_i Max \cup C_i Min)$.
- without loss of generality, $C_i Min = \emptyset$ when $i \equiv 1 \pmod{2}$ and $C_i Max = \emptyset$ when $i \equiv 0 \pmod{2}$.

We call these intervals max and min intervals. It is enough to hold the newest token $tkn$ on an event $a$ in a max interval. Suppose some event $a_{i+j}, j \geq 1$ occurs at time $t^*$. Let $t$ be the time $tkn$ reached $a_i$, and let $t' < t$ be the time some other token $tkn'$ reached $a_i$. If $tkn'$ satisfies the relevant constraints and can spawn a new token on $a_{i+j}$, then $tkn$ is able to do so as well. This is because its max constraints are less restrictive than any older token; for any max constraint $(a_i, a_{i+j}, w), t^* - t' < w \Rightarrow t^* - t < w$. Furthermore, there are no min constraints to check $tkn$ against, since any min constraint is either from an earlier interval and thus had been satisfied, or from a later interval, and thus its source is yet to be reached. Similarly, it is enough to hold the oldest token $tkn$ on an event $a$ in a min interval, while $tkn$ is not obsolete.

Therefore, for an IMM temporal pattern we need maintain only $O(n)$ graph tokens at any point of time in the execution.

### 3.3 History Tree

Next we show a way to improve space efficiency for any scheme in which we wish to keep a history for tokens, in particular when the history scheme is used. We use a shared data structure for tokens with intersecting histories. We maintain a history tree, which is a rooted tree data structure. Each node is a tuple $(a, t, \text{father})$, where $(a, t)$ is a timed event, and $\text{father}$ points to the father node. The root points to $\text{null}$.

Each token $tkn$ holds a pointer $tkn.h$ to a node in some history tree. When the first pattern event $a_1$ occurs at time $t_0$, a new history tree is generated with the root $(a_1, t_0, \text{null})$. Also, the new token that is created on event $a_1$ in the tracking automaton is given a pointer to the root. Suppose an event $a_{i+1}$ occurs at time $t$, so that a token $tkn$ on $a_i$ spawns a new token $tkn'$ on $a_{i+1}$. Then the node $te = (a_{i+1}, t, tkn.h)$ is added to the tree and we set $tkn'.h = te$. An illustration is shown in Fig. 8.
To obtain the history of some token tkn, we travel from tkn.h up to the root of the tree. Specifically if a token reaches the last event signifying a pattern match, we can reconstruct its history this way.

Let \( l \) be the number of graph tokens at some point of time in the execution. If we consider the history scheme, then where before we used \( O(l \cdot n) \) space to keep the tokens’ histories, now we use \( O(l + n) \) space per rooted tree. Each rooted tree is initially generated when the first event of the pattern occurs; all the subsequent descendant tokens spawned from this token will use the same rooted tree. If we have \( q \) rooted trees, then the tokens use \( O(q(l + n)) \) space in total; in particular if \( q = O(1) \), then the tokens use \( O(l + n) \) space in total.

We note that in the deadline scheme, when there are few (\( O(1) \)) overpassing edges in the pattern graph, the added cost of maintaining a history for the tokens is only \( O(l + n) \) as shown above. Therefore, if we wish to supply the history of a pattern match, the added cost is relatively small.

4 Concluding Remarks

In this work we introduced a novel framework for monitoring real-time systems for undesired behavior, based upon specifications given as temporal patterns. We devised a process for finding the closure of the temporal constraints semantics, and provided different schemes for on-line detection of temporal patterns.

A summary of complexity measures for handling different pattern types and schemes is shown in (Table 1). The measures are the number of comparisons per token, the space occupied by a token’s data structures, and the maximum

\[^4\] #graph tokens for sporadic temporal patterns is \( O(n) \) for IMM temporal patterns rather than \( O(k^n) \)
number of tokens on the graph at any point in time. We analyze the history scheme for many (up to $O(n^2)$) graph edges, and the deadline scheme for few ($O(1)$) graph edges. We distinguish between detection of continuous and sporadic patterns.

### Table 1. Tokens’ Resources and Costs

| pattern type | scheme(#edges) | measure | max constraints | max & min constraints |
|--------------|----------------|---------|-----------------|-----------------------|
| continuous   | history($O(n^2)$) | #comparisons | $O(n^2)$ | $O(n^2)$ |
|              |                 | space     | $O(n)$   | $O(n)$   |
|              |                 | #graph tokens | $O(n)$ | $O(n)$ |
| continuous   | deadline($O(1)$) | #comparisons | $O(n)$   | $O(n)$   |
|              |                 | space     | $O(1)$   | $O(1)$   |
|              |                 | #graph tokens | $O(n)$ | $O(n)$ |
| sporadic     | history($O(n^2)$) | #comparisons | $O(n^2)$ | $O(n^2)$ |
|              |                 | space     | $O(n)$   | $O(n)$   |
|              |                 | #graph tokens | $O(n)$ | $O(n)$ |
| sporadic     | deadline($O(1)$) | #comparisons | $O(n)$   | $O(n)$   |
|              |                 | space     | $O(1)$   | $O(1)$   |
|              |                 | #graph tokens | $O(n)$ | $O(k^n)$ |
| sporadic     | any scheme using a history tree | space used by q history trees and l tokens | $O(q(l+n))$ | $O(q(l+n))$ |

Specifications of properties for on-line verification may be more complex. Hence, another line of research may be broadening the input scope to temporal patterns as boolean formulae, where constraints are variables and histories are assignments. For example, the formula $((a, b, max, 10) \land (b, c, max, 15) \land (a, c, min, 20))$ evaluates to True under $((a, 0), (b, 8), (c, 21))$, and False under $((a, 0), (b, 8), (c, 18))$. The so-called regular temporal patterns we handled in this paper are in fact boolean formulae with $\land$ as the only connective. We say $H$ fits a temporal pattern formula if it evaluates to True under $H$. We define the language $TP-SAT$ as all temporal pattern formulae $TP$, such that there exists a history $H$ that fits $TP$. $TP-SAT \in NP$, and it is easy to show by a reduction from SAT that $TP-SAT$ is NP-Hard. However, some families of formulae may be tractable. DNF formulae for example may be seen as a collection of regular temporal patterns with the same sequence of events (albeit different temporal constraints), and handled accordingly.
References

1. Alur, R., Courcoubetis, C., Dill, D.: Model-Checking in Dense Real-Time (1993), *Information and Computation*, 104 (1), pp. 2-34.
2. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic Model checking: 10^{20} States and beyond (1992), *Information and Computation*, 98 (2), pp. 142-170.
3. Brukman, O., Dolev S.: Recovery Oriented Programming, *Proc. of the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems* (SSS 2006), LNCS 4280, pp. 152-168, 2006. Also a poster (with Shiman, M.,) in the *ACM 20th Symposium on Operating Systems Principles*, (SOSP), 2005. Journal version Recovery Oriented Programming: Runtime Monitoring of Safety and Liveness, *STTT International Journal on Software Tools for Technology*, accepted, April 2011.
4. Brukman, O., Dolev, S.: Self-* Programming Run-Time Parallel Control Search for Reflection-Box, *Proc. of the 6th NASA Langley Formal Methods Workshop*, April 2008. A poster in the *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, (SASO) 2008, and in *Automata: from Mathematics to Applications*, (AutoMathA) 2009.
5. Brukman, O., Dolev, S., Kolodner, E.: Self-Stabilizing Autonomic Recoverer for Eventual Byzantine Software, *Journal of Systems and Software*, 81, pp. 2315-2327, 2008.
6. Crochemore, M.: String Matching with Constraints, *Proc. MFCS’88 Symp. Lecture Notes in Computer Science*, Vol. 324 (Springer, Berlin, 1988) 44-58.
7. Clarke, E.M., Grumberg, O., Peled D.A.: Model Checking, MIT Press, 1999.
8. Dechter, R., Meiri, I., Pearl, J.: Temporal Constraint Networks, *Artificial Intelligence*, 49:61-95, 1991.
9. Dolev, S., Stomp, F.: Safety Assurance via On-Line Monitoring, *Distributed Computing*, Volume 16, Number 4, pp. 269-277, December 2003.
10. Laroussinie, f., Markey, N., Schnoebelen, Ph.: Efficient Timed Model Checking for Discrete-Time Systems, *Theor. Comput. Sci.* 353, 1 (March 2006), 249-271.
11. Leiserson, C.E., Saxe, J.B.: A Mixed-Integer Linear Programming Problem which is Efficiently Solvable, in *Proceedings 21st Annual Allerton Conference on Communications, Control, and Computing* (1983) 204-213.
12. Liao, Y.Z., Wong, C.K.: An algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems* 2 (2) (1983) 62-69.
13. Zeigler, B., Kim, T.G., Prachhofer, H.: Theory of Modeling and Simulation. Second edition. Academic Press, New York, 2000.
Appendix

A Proposition proofs

Proof (of Prop. 1).
Equivalence: We define a backward path from b to a as a path that consists only of backward 0-edges that leads from b to a; denote such a path by $BP(b, a)$. We define a full order $\preceq$ on the events of the pattern, so that $a \preceq b$ iff a appears chronologically before b. For simplicity, we break the updating of G to n single update steps with a single update step at a time. We show that $G \equiv G_0 \equiv G_1 \equiv \ldots \equiv G_n = G'$, where these graphs are the intermediate graphs along the n update steps. We examine the update step of $G_i$. If an edge $e = (u, v, w)$ is removed from $G_i$, then obviously $G_i \preceq G_{i+1}$. Let $P$ be the shortest $(u, v)$-path. We define $b_0 = u, a_k = v$. The general form of this path is $(BP(b_0, a_0), (a_0, b_1), BP(b_1, a_1), (a_1, b_2), \ldots, (a_{k-1}, b_k), BP(b_k, a_k))$. There is an edge $(a_i, b_{i+1})$ (either simple or overpassing) for each $0 \leq i \leq k - 1$. For each $1 < i < k$ it must hold that $a_{i-1} \preceq a_i$, else we can remove the sub-path $(a_{i-1}, b_i, BP(b_i, a_{i-1}))$ from P and get a shorter $(u, v)$-path, a contradiction. For each $0 < i < k - 1$ it must hold that $b_i \preceq b_{i+1}$, else we can remove the sub-path $(BP(b_{i+1}, a_i), a_i, b_{i+1})$ from P and get a shorter $(u, v)$-path, a contradiction. Also, either $a_i \preceq b_i$ or $a_i = b_i$. We note that a simple path P from u to v is a special case of this form. Also, $w(P) = \sum w(a_i, b_{i+1}) < w(e)$. Let H be a history that fits $G_{(i+1)}$. Then $H(u, v) \leq H(a_0, b_k) = H(a_0, b_1) + \sum H(b_i, b_{i+1}) \leq \sum H(a_i, b_{i+1}) \leq \sum w(a_i, b_{i+1}) = w(P) < w(e)$. Since the removal of e is the only change in the graph, H must also fit $G_i$. We get that $G_{i+1} \preceq G_i$, and therefore $G_i \equiv G_{i+1}$.

If the weight of a simple edge $e = (u, v, w)$ in $G_i$ is reduced to $w'$ in $G_{(i+1)}$, then there is an edge $e' = (u', v')$ with weight $w'$ that overpasses $e$, and clearly $G_{(i+1)} \preceq G_i$. Let $P$ be the shortest $(u, v)$-path. This path necessarily takes the following form: $P = (BP(u, u'), (u', v'), BP(v', v))$. It may hold that $u' = u$ or $v' = v$ (but not both). Let H be a history that fits $G_i$. Then $H(u, v) \leq H(u', v') \leq w'$. Since the reduced weight of e is the only change in the graph, H must also fit $G_{i+1}$. We get that $G_i \preceq G_{i+1}$, and therefore $G_i \equiv G_{i+1}$. We thus showed that for every $1 < i < n - 1$, $G_i \equiv G_{(i+1)}$, and hence $G \equiv G'$. Minimality: Assume toward a contradiction that we can remove an overpassing edge $e = (a, b)$ from $G'$ or reduce the weight of a simple edge $e = (a, b)$ in $G'$, such that the resulting graph $G''$ is equivalent to $G'$. Assume $e$ is a simple edge whose weight is reduced from $w'$ in $G'$ to $w''$ in $G''$. $G'$ is the graph generated by the preprocessing process, so we know that $e$ is the shortest $(a, b)$-path in $G'$. Let H be a history that fits $G'$ and uses the full weight of $e$ in $G'$, namely the time elapsed between the events $a$ and $b$ is $w'$. We are assured such a history exists because e is the shortest $(a, b)$-path in $G'$. the time elapsed between events $a$ and $b$ cannot violate any overpassing edge $e^*$ since it must weigh at least as much as $e$, and we can assign all other simple edges $0$
time lapses in $H$. $H$ clearly does not fit $G''$ since it violates the constraint on $e$ in $G''$. This is contradictory to the equivalence of $G'$ and $G''$.

Assume $e$ is an overpassing edge that is removed. $G'$ is the graph generated in the preprocessing phase, so we know that $e$ is the shortest $(a, b)$-path. Next we show that there is a history $H'$ that fits $G'$ (and therefore $G''$) such that $H(a, b) = w(e)$, and that $H'$ can be augmented by adding 1 to $H'(a, b)$ resulting in a new history $H''$, so that $H''$ fits $G''$, but not $G'$. Let $(e_1, e_2, ..., e_k)$ be the simple $(a, b)$-path. We describe a recursive method of constructing $H'$. At first $H'(a_i, a_{i+1}) = 0$ for all $1 \leq i \leq n - 1$. In particular $H'(e_i) = 0$ for all $1 \leq i \leq k - 1$. Obviously $H'$ fits $G'$. Let $H'(e_i) = x_i$ for all $1 \leq i \leq k - 1$ at some step of the recursive method. We show that while $\sum_{j=1}^{k} x_i < w(e)$, we can add 1 to some $x_i$ and remain with a history that fits $G'$. We say $e_i$ is full if $x_i = w(e_i)$. The edges $e_1, ..., e_k$ cannot be all full, because then the $(a, b)$-path $(e_1, ..., e_k)$ is shorter than $e$, which is a contradiction. Now, there has to be some $x_i$ that can be augmented to $x_i + 1$ so that we remain with a history that fits $G'$. 

Explanation: Assume to the contrary that such $x_i$ does not exist. Then every $e_j$ that is not full is overpassed by an edge $e'_j$ such that $w(e'_j) = x_j$. In this case, we can construct an $(a, b)$-path $P$ as follows: We advance from $a$ towards $b$ using full edges. When we arrive at an edge $e_j$ that is not full, we may travel back with 0-edges to the source of $e'_j$, add $e'_j$ to $P$ thus overpassing $e_j$, and continue advancing by the same rules until we reach $b$. Clearly $w(P) = \sum_{j=1}^{k} x_i < w(e)$, which is a contradiction. Hence there exists the desired $e_i$ that is not full, where $x_i$ can be augmented to $x_i + 1$ so that we remain with a history that fits $G'$.

This recursive method is finite, for at some point we shall have values $x_1, ..., x_k$ such that $H'(a_i, b) = \sum_{j=1}^{k} x_i = w(e)$ and $H'$ fits $G'$. There has to be some $e_i$ that is not full in $H'$ because $e$ is the unique shortest $(a, b)$-path (for any other $(a, b)$-path $P$, $w(P) > w(e)$). By the same explanation as above, there has to be some $e_i$ such that $x_i$ can be augmented to $x_i + 1$ so that we remain with a history that fits $G''$, since otherwise we get an $(a, b)$-path $P$ in $G''$, with $w(P) = w(e)$. We denote the history following this augmentation $H''$. $H''$ fits $G''$ but does not fit $G'$, and therefore $G' \neq G''$, which is a contradiction. 

Proof (of Prop. 4):

Equivalence: First, it is easy to observe that $TP' \leq TP$: Let $H$ be a pattern history. For any two pattern events $a, b$ such that $a \leq b$, there is a simple $(a, b)$-path $P$ of simple edges in $G$, and the weight of the shortest $(a, b)$-path $P'$ must be at most $w(P)$. We get: $H$ fits $TP' \Rightarrow H(a, b) \leq w(P') \Rightarrow H(a, b) \leq w(P)$. Similarly, there is a simple backward $(b, a)$-path $Q$ of simple backward edges in $G$, and the weight of the shortest $(b, a)$-path $Q'$ must be at most $w(Q)$. We get: $H$ fits $TP' \Rightarrow H(a, b) \geq -w(Q') \Rightarrow H(a, b) \geq -w(Q)$. This holds for all pairs of events $a, b$ such that $a \leq b$, hence $TP' \leq TP$. If $P$ is a path between two events in $G$, we define $L(P)$ as the number of edges in $P$.

We prove by induction that $TP \leq TP'$. The induction statement is as follows: For any $k \in \mathbb{N}$, for any pair of pattern events $a, b$ such that $a \leq b$, the following hold: (a) If $P$ is an $(a, b)$-path with $L(P) = k$ in $G$, then in any history
$H$ that fits $TP$, $H(a,b) \leq w(P)$ holds. (b) If $Q$ is a $(b,a)$-path $L(Q) = k$ in $G$, then in any history $H$ that fits $TP$, $H(a,b) \geq -w(Q)$ holds.

If (a) and (b) hold and $H$ is some history that fits $TP$, then it also fits $TP'$ because its constraints are defined by the weights of paths in $G$.

**Basis.** If $P$ is an $(a,b)$-path with weight $w$ and with $L(P) = 1$ in $G$, then $P$ is a single edge between the events $a,b$, i.e. it represents a max constraint $w$. If $H$ is a history that fits $TP$ then $H(a,b) \leq w$, and thus (a) holds.

Similarly, if $Q$ is a $(b,a)$-path with weight $w$ and with $L(Q) = 1$, then $Q$ is a single backward edge between the events $b,a$, i.e. it represents a min constraint $-w$. If $H$ is a history that fits $TP$ then $H(a,b) \geq -w$, and thus (b) holds.

**Inductive Step.** We assume the inductive hypothesis, namely that the induction step is true for some $k \in \mathbb{N}$, and prove that it is then also true for $k+1$. Let $H$ be some history that fits $TP$:

(a) Let $P$ be an $(a,b)$-path $G$ with $L(P) = k + 1$, and let $H$ be some history that fits $TP$. Let $P_0$ be the $k$-prefix of $P$, i.e. $P = (P_0, (a',b,w))$ for some $(a',b,w) \in G$. We consider three distinct cases:

- $a' \leq a$: Here $(a',b,w) \in CMax$. By the inductive hypothesis part (b), $H(a',a) \geq -w(P_0)$. Also, due to the edge $(a',b,w)$, $H(a',b) \leq w$ holds. Hence $H(a,b) \leq w - (w(P_0)) = w + w(P_0) = w(P)$.

- $a \leq a'$: Here too $(a',b,w) \in CMax$. By the inductive hypothesis part (a), $H(a,a') \leq w(P_0)$. Also, due to the edge $(a',b,w)$, $H(a',b) \leq w$ holds. Hence $H(a,b) \leq w + w(P_0) = w(P)$.

(b) Let $Q$ be a $(b,a)$-path in $G$ with $L(Q) = k + 1$, and let $H$ be some history that fits $TP$. Let $Q_0$ be the $k$-prefix of $Q$, i.e. $Q = (Q_0, (b',a,w))$ for some $(b',a,w) \in G$. We consider three distinct cases:

- $b \leq b'$: Here $(a,b',-w) \in CMin$. By the inductive hypothesis part (a), $H(b,b') \leq w(Q_0)$. Also, due to the edge $(b',a,w)$, $H(a,b') \geq -w$ holds. Hence $H(a,b) \geq -w - w(Q_0) = -(w + w(Q_0)) = -w(Q)$.

- $a \leq b'$: Here too $(a,b',-w) \in CMin$. By the inductive hypothesis part (b), $H(b',b) \geq -w(Q_0)$. Also, due to the edge $(b',a,w)$, $H(a,b') \geq -w$ holds. Hence $H(a,b) \geq -w(Q_0) - w = -w(Q)$.

No-Negative: Assume there is a history $H$ that fits $TP$. Now, if there is a negative cycle $C$ in $G$, let $a,b$ be events in $C$ such that $a \leq b$. Then we may describe $C$ as the concatenation of two paths, $C = (P_{a,b}, P_{b,a})$, where $P_{a,b}$ is an $(a,b)$-path and $P_{b,a}$ is a $(b,a)$-path. From (1) we know that $H(a,b) \leq w(P_{a,b})$ and $H(a,b) \geq -w(P_{b,a})$ hold. But $0 > w(C) = w(P_{a,b}) + w(P_{b,a})$, and hence $H(a,b) \geq -w(P_{b,a}) + w(P_{a,b}) \geq H(a,b)$, a contradiction. Therefore there is no negative cycle in $G$. 

Now assume there is no negative cycle in $G$. We prove inductively that there exists a history that fits $TP$. We define the $k$-prefix of $TP$, denoted as $TP_k$, as the temporal pattern defined by the events $a_1, \ldots, a_k$ and by the time windows $\{tw \mid tw \in TP \land ((dst(tw) \leq a_k) \lor (dst(tw) = a_k))\}$. The induction statement is as follows: For any $k \in \mathbb{N}$, $2 \leq k \leq n$, there exists a history that fits $TP_k$.

**Basis.** $k = 2$, $\overline{tw}(a_1, a_2) \leq tw(a_1, a_2)$, else there is an $(a_1, a_2)$-path and an $(a_2, a_1)$-path such that their sum is negative, i.e. a negative cycle, in contradiction to our assumption. A history $((a_1, 0), (a_2, tw(a_1, a_2)))$ clearly fits $TP_2$.

**Inductive Step.** We assume the inductive hypothesis, namely that the induction step is true for some $k \in \mathbb{N}$, $2 \leq k \leq n - 1$, and prove that it is then also true for $k+1$. Let $H$ be a history that fits $TP_k$. Let $dl = \min_{i=3}^{k+2} \{H(a_i) + tw(a_i, a_{k+1})\}$. If $dl < ll$, let $H = (a_k, a_{k+1})$ such that $dl \leq H(a_{k+1}) \leq dl$, and $H$ will fit $TP_{k+1}$.

We assume toward a contradiction that $dl < ll$. Let $a_i$ be an event such that $H(a_i) + tw(a_i, a_{k+1}) = ll$, and let $a_j$ be an event such that $H(a_j) + tw(a_j, a_{k+1}) = dl$. Let $P_{k+1,i}$ and $P_{j,k+1}$ be paths in $G$ with weights $-tw(a_i, a_{k+1})$ and $tw(a_j, a_{k+1})$, respectively. We consider two cases.

- $(a_j \leq a_i) \lor (a_j = a_i)$: Let $P_{i,j}$ be the simple backward $(a_i, a_j)$-path (empty if $a_j = a_i$). $w(P_{i,j}) \leq 0$. We get a cycle $C = (P_{j,k+1}, P_{k+1,i}, P_{i,j})$. $w(C) = w(P_{j,k+1}) + w(P_{k+1,i}) + w(P_{i,j}) \leq H(a_{k+1})$ such that $ll - H(a_{k+1}) < dl$. $C$ is a negative cycle, in contradiction to the assumption that there are no negative cycles in $G$.

- $(a_i \leq a_j)$: The path $(P_{j,k+1} \cup P_{k+1,i})$ has weight $\overline{tw}(a_j, a_{k+1}) - tw(a_i, a_{k+1})$, hence $-tw(a_i, a_j) \leq \overline{tw}(a_j, a_{k+1}) - tw(a_i, a_{k+1})$, and therefore $tw(a_i, a_{k+1}) - tw(a_j, a_{k+1}) \leq tw(a_i, a_j)$. Now $H(a_j) - H(a_i) = (dl - \overline{tw}(a_j, a_{k+1})) - dl \leq tw(a_i, a_{k+1}) \leq tw(a_i, a_{k+1})$. This is contradictory to the assumption that $H$ fits $TP_k$.

Minimality: Let $tw(a, b)$ be a time window in $TP'$. We show that any contraction from above or from below, i.e. decreasing $\overline{tw}(a, b)$ or increasing $tw(a, b)$ respectively, will result in a temporal pattern $TP''$ that is not equivalent to $TP'$. Let $H$ be some history that fits $TP'$. If there are $k - 1$ events between $a$ and $b$, let $tw_1, tw_2, \ldots, tw_k$ be the simple time windows between $a$ and $b$. Let the source and destination of $tw_i$ be $b_i$ and $b_{i+1}$ respectively, $1 \leq i \leq k$. Specifically $a = b_1$ and $b = b_{k+1}$.

Assume $tw(a, b)$ is decreased. We describe a recursive method of augmenting $H$. Let $H(tw_i) = x_i$ for all $1 \leq i \leq k$ at some step of the recursive method. We show that while $\sum_{i=1}^{k} x_i \leq tw(a, b)$, we can add 1 to some $x_i$ and remain with a history that fits $TP'$. We say $tw_i$ is full if $x_i = tw_i$. $tw_1, \ldots, tw_k$ cannot be all full, because then the weight of the $(a, b)$-path $(P_1, \ldots, P_k)$ in $TP$, where $P_i$ is the shortest $(b_i, b_{i+1})$-path in $TP$, is smaller than $\overline{tw}(a, b)$, which is a contradiction to the definition of $tw$. Now, there has to be some $x_i$ that can be augmented to $x_i + 1$ so that we remain with a history that fits $TP'$. 
Explanation: Assume to the contrary that such \( x_i \) does not exist. Then every \( tw_j \) that is not full is subsumed by a time window \( tw'_j \) such that \( tw'_j = x_j \), i.e. there is a path \( P'_j \) between the source and destination of \( tw'_j \), with \( w(P'_j) = x_j \).

In this case, we can construct an \((a,b)\)-path \( P \) as follows: We advance from \( a \) towards \( b \) using full time windows. When we arrive at a time window \( tw_j \) that is not full, we may travel back with non-positive edges to the source of \( tw'_j \), add \( P'_j \) to \( P \) thus overpassing \( b_{j+1} \), and continue advancing by the same rules until we reach \( b \). Clearly \( w(P) \leq \sum_{i=1}^{k} x_i < tw(a,b) \), which is a contradiction to the definition of \( tw \).

Hence there exists the desired \( tw_i \) that is not full, where \( x_i \) can be augmented to \( x_i + 1 \) so that we remain with a history that fits \( TP' \). This procedure is finite, for at some point we shall have values \( x_1, ..., x_k \) such that \( H(a,b) = \sum_{i=1}^{k} x_i = \overline{tw}(a,b) \) and \( H \) fits \( TP' \). Let \( TP'' \) be the temporal pattern after decreasing \( \overline{tw}(a,b) \). Clearly \( H \) does not fit \( TP'' \), hence \( TP' \neq TP'' \).

The proof of the case where \( tw(a,b) \) is increased is similar to (a). \( \square \)