Foundations of Dispatchability for Simple Temporal Networks with Uncertainty

Luke Hunsberger[1] [a] and Roberto Posenato[2] [b]

[1] *Computer Science Department, Vassar College, Poughkeepsie, NY, U.S.A.*

[2] *Dipartimento di Informatica, Universitàdegli Studi di Verona, Verona, Italy*

Keywords: Planning and Scheduling, Temporal Constraint Networks, Dispatchability, Real-Time Execution.

Abstract: Simple Temporal Networks (STNs) are a widely used formalism for representing and reasoning about temporal constraints on activities. The dispatchability of an STN was originally defined as a guarantee that a specific real-time execution algorithm would necessarily satisfy all of the STN's constraints while preserving maximum flexibility but requiring minimal computation. A Simple Temporal Network with Uncertainty (STNU) augments an STN to accommodate actions with uncertain durations. However, the dispatchability of an STNU was defined differently: in terms of the dispatchability of its so-called STN projections. It was then argued informally that this definition provided a similar real-time execution guarantee, but without specifying the execution algorithm. This paper formally defines a real-time execution algorithm for STNUs that similarly preserves maximum flexibility while requiring minimal computation. It then proves that an STNU is dispatchable if and only if every run of that real-time execution algorithm necessarily satisfies the STNU's constraints no matter how the uncertain durations play out. By formally connecting STNU dispatchability to an explicit real-time execution algorithm, the paper fills in important elements of the foundations of the dispatchability of STNUs.

1 INTRODUCTION

Temporal networks are formalisms for representing and reasoning about temporal constraints on activities. Many kinds of temporal networks differ in the kinds of constraints and uncertainty that they can accommodate. Typically, the more expressive the network, the more expensive the corresponding computational tasks.

Simple Temporal Networks (STNs) are the most basic and most widely used kind of temporal network (Dechter et al., 1991). An STN can represent deadlines, release times, duration constraints, and inter-action constraints. The basic computational tasks associated with STNs can be done in polynomial time. An STN is *consistent* if it has a solution (as a constraint-satisfaction problem). But, imposing a fixed solution in advance of execution (i.e., before any actions are actually performed) is often unnecessarily inflexible. Instead, it can be desirable to postpone, as much as possible, decisions about the precise timing of actions to allow an executor to react to unexpected events without having to do expensive replanning. In other words, it can be desirable to take advantage of the inherent flexibility afforded by the STN representation. However, postponing execution decisions invariably requires real-time computations to, for example, propagate the effects of such decisions throughout the network. An effective real-time execution algorithm, responsible for saying when actions should be done, must therefore limit the amount of real-time computation. A Real-Time Execution (RTE) algorithm that preserves maximum flexibility while requiring minimal computation has been presented for STNs (Muscettola et al., 1998). Unfortunately, the RTE algorithm does not necessarily successfully execute all consistent STNs (i.e., it does not guarantee the satisfaction of all of the STN's constraints). However, it has been shown that every consistent STN can be converted into an *equivalent* network that the RTE algorithm *will* necessarily successfully execute—no matter how the algorithm chooses to exploit the network's flexibility (Muscettola et al., 1998). Such networks are called *dispatchable*. They provide applications with both flexibility *and* compu-

[a] https://orcid.org/0009-0005-8603-4803

[b] https://orcid.org/0000-0003-0944-0419

Hunsberger, L. and Posenato, R.
Foundations of Dispatchability for Simple Temporal Networks with Uncertainty.
DOI: 10.5220/0012360000003636

In *Proceedings of the 16th International Conference on Agents and Artificial Intelligence (ICAART 2024) - Volume 2*, pages 253-263
ISBN: 978-989-758-680-4; ISSN: 2184-433X

tational efficiency.

Simple Temporal Networks with Uncertainty (STNUs) augment STNs to accommodate actions with uncertain durations (Morris et al., 2001). Although more expressive than STNs, the basic computational task associated with STNUs can also be done in polynomial time (Morris, 2014; Cairo et al., 2018). An STNU is *dynamically controllable* (DC) if there exists a dynamic strategy for executing its actions such that all of its constraints will be satisfied no matter how the uncertain action durations play out—within their specified bounds. An execution strategy is *dynamic* in that it can *react* to observations of action durations as they occur. Unlike solutions for consistent STNs, dynamic strategies for DC STNUs typically require exponential space and thus cannot be computed in advance. Instead, *the relevant portions* of such strategies can be computed incrementally, during execution. As with STNs, it is important to preserve maximal flexibility while requiring minimal computation during execution. Hence, the notion of dispatchability has also been defined for STNUs (Morris, 2014). However, unlike for STNs, the dispatchability of an STNU was not specified as a constraint-satisfaction guarantee for a particular real-time execution algorithm, but instead in terms of the dispatchability of its STN *projections*. (A projection of an STNU is the STN that results from assigning a fixed duration to each action.) Since STN dispatchability can be checked by analyzing the associated STN graph (Morris, 2016), this definition is attractive. However, it was only argued informally that dispatchability for an STNU, defined in this way, would provide a similar constraint-satisfaction guarantee in the context of real-time execution. Nonetheless, polynomial algorithms for converting DC STNUs into equivalent dispatchable networks have been presented (Morris, 2014; Hunsberger and Posenato, 2023).

Since the primary motivation for dispatchability is to provide a real-time execution guarantee, it is important to formally connect STNU dispatchability to a real-time execution algorithm. This paper provides such a connection. First, it defines a real-time execution algorithm for STNUs, called RTE*, that preserves maximal flexibility while requiring minimal computation. Then it proves that an STNU is dispatchable if and only if every run of the RTE* algorithm necessarily satisfies its constraints, no matter how the uncertain durations turn out. In this way, the paper fills an important gap in the foundations of STNU dispatchability.

The rest of the paper is organized as follows. Section 2 summarizes the main definitions and results for the dispatchability of Simple Temporal Networks (STNs). Section 3 reviews Simple Temporal Networks with Uncertainty (STNUs) and how the concept of dispatchability has been extended to them using Extended STNUs (ESTNUs). Section 4 introduces a real-time execution algorithm for ESTNUs, called RTE*, and proves its correctness. Section 5 summarizes the contributions of the paper and sketches possible future work.

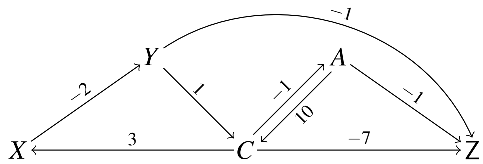


Figure 1: A sample STN graph.

2 STN DISPATCHABILITY

A Simple Temporal Network (STN) is a pair, $(\mathcal{T}, \mathcal{C})$, where $\mathcal{T}$ is a set of real-valued variables called *timepoints* (TPs) and $\mathcal{C}$ is a set of binary difference constraints, called *ordinary constraints,* each of the form $Y - X \leq \delta$, where $X, Y \in \mathcal{T}$ and $\delta \in \mathbb{R}$ (Dechter et al., 1991). Typically, we let $n = |\mathcal{T}|$ and $m = |\mathcal{C}|$. With no loss of generality, it is convenient to assume that each STN has a special timepoint Z whose value is fixed at *zero* (or some other convenient timestamp) and is constrained to occur at or before every other timepoint.[1] Each STN has a corresponding graph, $(\mathcal{T}, \mathcal{E})$, where the timepoints in $\mathcal{T}$ serve as nodes and each constraint $Y - X \leq \delta$ in $\mathcal{C}$ corresponds to a labeled directed edge $X \xrightarrow{\delta} Y$ in $\mathcal{E}$, called an *ordinary edge.* For convenience, such edges will be notated as (X, δ, Y). Figure 1 shows a sample STN graph. An STN is *consistent* if it has a solution as a constraint satisfaction problem. An STN is consistent if and only if its graph has no negative cycles (Dechter et al., 1991).

Although checking the consistency of an STN is important and can be done in polynomial time, fixing a solution in advance undermines the inherent flexibility of the STN representation. Instead, it can be desirable to preserve as much flexibility as possible until actions are actually performed (i.e., during the "real-time execution"), while minimizing real-time computation.

Toward that end, consider the Real-Time Execution (RTE) algorithm for STNs given in Algo-

[1] It is not hard to show that in any consistent STN (see below) there is at least one TP that can play the role of Z (i.e., constrained to occur at or before every other TP).

Algorithm 1: RTE: real-time execution for STNs.

Input: $(\mathcal{T}, \mathcal{C})$, an STN with graph $(\mathcal{T}, \mathcal{E})$
Output: A function, $f\colon \mathcal{T} \to [0,\infty)$ or `fail`

1 **foreach** $X \in \mathcal{T}$ **do**
2 $\quad \mathrm{TW}(X) = [0,\infty)$
3 $\mathcal{U} := \mathcal{T};\ \mathtt{now} = 0$
4 $\mathtt{Enabs} := \{X \in \mathcal{T} \mid X$ has no outgoing negative edges$\}$
5 **while** $\mathcal{U} \neq \{\}$ **do**
6 $\quad$ **if** $\mathtt{Enabs} = \emptyset$ **then**
7 $\quad\quad$ **return** `fail`
8 $\quad \ell := \min\{lb(W) \mid W \in \mathtt{Enabs}\}$
9 $\quad u := \min\{ub(W) \mid W \in \mathtt{Enabs}\}$
10 $\quad$ **if** $[\ell,u] \cap [\mathtt{now},\infty) = \emptyset$ **then**
11 $\quad\quad$ **return** `fail`
12 $\quad$ Select any $X \in \mathtt{Enabs} \mid \mathtt{TW}(X) \cap [\mathtt{now},u] \neq \emptyset$
13 $\quad$ Select any $t \in \mathtt{TW}(X) \cap [\mathtt{now},u]$
14 $\quad$ Remove X from $\mathcal{U}$
15 $\quad f(X) := t;\ \mathtt{now} := t$
16 $\quad$ Propagate $f(X) = t$ to X's neighbors in $\mathcal{E}$
17 $\quad \mathtt{Enabs} := \{Y \in \mathcal{U} \mid$ all negative edges from Y terminate at TPs not in $\mathcal{U}\}$
18 **return** f

rithm 1 (Muscettola et al., 1998).[2] It provides maximum flexibility by maintaining for each timepoint X a *time window* $\mathrm{TW}(X)$ (initially $[0,\infty)$, Line 2), and providing freedom for which timepoint to execute next and when to execute it (Lines 8 to 13). To minimize real-time computation, the effects of each execution decision, $X = t$ (represented in the pseudocode by setting $f(X) = t$ at Line 15) are propagated only *locally,* to the *neighbors* of X in the STN graph (i.e., the timepoints connected to X by a single edge) (Line 16).

After initializing the time windows (Line 2), the RTE algorithm initializes the *current time* `now` to 0 and the set $\mathcal{U}$ of *unexecuted* timepoints to $\mathcal{T}$ (Line 3); and then the set of *enabled* timepoints to those having no outgoing negative edges (Line 4). (A timepoint Y is enabled for execution if it is *not* constrained to occur *after* any *unexecuted* timepoint—equivalently, if there are no *negative* edges from Y to any *unexecuted* timepoint.) Each iteration of the **while** loop (Lines 5 to 17) begins by computing the interval $[\ell,u]$, where ℓ is the minimum lower bound of the time windows among the enabled timepoints (i.e., the earliest time at which something *could* happen) and u is the minimum upper bound among those same time windows (i.e., the deadline by which something *must* happen) (Lines 8 and 9).[3] The algorithm fails if that interval does not include times at or after `now` (Line 10). Next (Line 12), it selects one of the enabled timepoints X whose time window $\mathtt{TW}(X)$ has a non-empty intersection with $[\mathtt{now},u]$, and then (Line 13) selects any time $t \in \mathtt{TW}(X) \cap [\mathtt{now},u]$ at which to execute it. (If $[\ell,u] \cap [\mathtt{now},\infty)$ is non-empty, then there must be such an X.) After assigning X to t (Line 15), it then propagates the effects of that assignment to X's neighbors in the STN graph (Line 16). In particular, for any non-negative edge $(X,\delta,V) \in \mathcal{E}$, it updates the time window for V as follows: $\mathrm{TW}(V) := \mathrm{TW}(V) \cap (-\infty, t+\delta]$. Similarly, for each negative edge $(U,-\gamma,X)$, it updates U's time window: $\mathrm{TW}(U) := \mathrm{TW}(U) \cap [t+\gamma,\infty)$. Finally, it updates the set of enabled timepoints (Line 17) in preparation for the next iteration.

The RTE algorithm for STNs provides maximal flexibility in that any solution to a consistent STN can be generated by an appropriate sequence of choices at Lines 12 to 13. In addition, it requires minimal computation by performing only *local* propagation (at Line 16). However, it does *not* provide a constraint-satisfaction guarantee for *all* runs on consistent STNs, as illustrated by the sample run-through of the algorithm shown in Table 1(a), which motivates the work on STN *dispatchability,* as follows.

Definition 1 (Dispatchability Muscettola et al. (1998))**.** *An STN $\mathcal{S} = (\mathcal{T}, \mathcal{C})$ is* dispatchable *if every run of the RTE algorithm (Algorithm 1) on the corresponding STN graph $\mathcal{G} = (\mathcal{T}, \mathcal{E})$ necessarily generates a solution for $\mathcal{S}$.*

Muscettola et al. (1998) showed that for consistent STNs, the all-pairs, shortest-paths (APSP) graph is necessarily dispatchable, but its $O(n^2)$ edges cancel the benefits of *local* propagation. Their $O(n^3)$-time *edge-filtering* algorithm computes an equivalent minimal dispatchable STN by starting with the APSP graph, then removing *dominated* edges (i.e., edges not needed for dispatchability). A faster $O(mn + n^2 \log n)$-time algorithm accumulates undominated edges without first building the APSP graph (Tsamardinos et al., 1998).

Morris (2016) later found a graphical characterization of STN dispatchability in terms of *vee-paths.*

Definition 2 (Vee-path (Morris, 2016))**.** *A* vee-path *comprises zero or more negative edges followed by zero or more non-negative edges.*

[2] Muscettola et al. (1998) refer to their algorithm as either the *Time Dispatching Algorithm* (TDA) or the *Dispatching Execution Controller* (DEC). The RTE algorithm presented here is equivalent, although organized somewhat differently and using different notation.

[3] In Algorithm 1, $\mathtt{lb}(X)$ and $\mathtt{ub}(X)$ respectively denote the lower and upper bounds from X's time window, $\mathtt{TW}(X)$.

Table 1: Sample runs of the RTE algorithm.
(a) A sample run of the RTE algorithm on the *consistent* STN from Figure 1.

Iter.	Enabs	$\mathtt{TW}(Z)$	$\mathtt{TW}(A)$	$\mathtt{TW}(C)$	$\mathtt{TW}(X)$	$\mathtt{TW}(Y)$	$[\ell, u]$	`now`	Exec.
Init.	$\{Z\}$	$[0,\infty]$	$[0,\infty)$	$[0,\infty)$	$[0,\infty)$	$[0,\infty)$	$[0,\infty)$	0	$Z := 0$
1	$\{A,Y\}$	—	$[1,\infty)$	$[7,\infty)$	$[0,\infty)$	$[1,\infty)$	$[0,\infty)$	0	$Y := 4$
2	$\{A,X\}$	—	$[1,\infty)$	$[\mathbf{7},\mathbf{5}]$	$[6,\infty)$	—	$[1,\infty)$	4	$A := 8$
3	$\{C,X\}$	—	—	$[\mathbf{9},\mathbf{5}]$	$[6,\infty)$	—	$[\mathbf{6},\mathbf{5}]$	**8**	**`fail`**

(b) A sample run of the RTE algorithm on the *dispatchable* STN from Figure 3.

Iter.	Enabs	$\mathtt{TW}(Z)$	$\mathtt{TW}(A)$	$\mathtt{TW}(C)$	$\mathtt{TW}(X)$	$\mathtt{TW}(Y)$	$[\ell, u]$	`now`	Exec.
Init.	$\{Z\}$	$[0,\infty)$	$[0,\infty)$	$[0,\infty)$	$[0,\infty)$	$[0,\infty)$	$[0,\infty)$	0	$Z := 0$
1	$\{A,Y\}$	—	$[1,\infty)$	$[7,\infty)$	$[0,\infty)$	$[6,\infty)$	$[1,\infty)$	0	$A := 8$
2	$\{C,Y\}$	—	—	$[9,18]$	$[0,\infty)$	$[8,\infty)$	$[8,18]$	8	$C := 15$
3	$\{Y\}$	—	—	—	$[0,18]$	$[8,16]$	$[8,16]$	15	$Y := 16$
4	$\{X\}$	—	—	—	$[18,18]$	—	$[18,18]$	16	$X := 18$

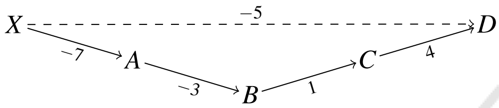


Figure 2: A sample *vee-path* that dominates a direct edge.

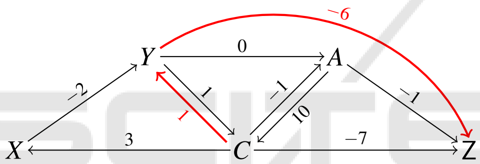


Figure 3: An equivalent dispatchable STN graph.

Figure 2 shows a sample vee-path from X to Y that dominates the (dashed) direct edge from X to Y. For this vee-path, the enablement condition (Line 12) ensures that the RTE algorithm will execute B before A, and A before X; hence, local propagation ensures the satisfaction of the edges $(X,-7,A)$ and $(A,-3,B)$. On the other side, if the algorithm executes C before B, then the edge $(B,1,C)$ is automatically satisfied; otherwise, local propagation ensures its satisfaction. Similarly, the RTE algorithm necessarily satisfies the edge $(C,4,D)$. Since the algorithm satisfies all the edges in the vee-path, it also satisfies the direct edge $(X,-5,Y)$. Hence that edge is not needed to ensure dispatchability.

Theorem 1 (Morris (2016))**.** *An STN is dispatchable iff for each path from any X to any Y in the STN graph, there is a* shortest *path from X to Y that is a* vee-path.

Figure 3 shows a dispatchable STN that is equivalent to the STN from Figure 1 (new edges are thick and red). It is easy to check that each path has a corresponding vee-path that is a shortest path. Table 1(b) shows a sample run of the RTE algorithm on this dispatchable STN, which necessarily generates a solution.

RTE Complexity. With appropriate data structures, the RTE algorithm can be implemented to run in $O(n^2)$ worst-case time, while allowing for maximum flexibility in the selection of the timepoint X to execute next and the time t at which to execute it. The local propagations involve m updates, each done in constant time. The set of enabled timepoints can be implemented by keeping, for each timepoint, a count of its outgoing negative edges. Whenever a negative edge is processed, the count for the source of that edge is decremented. When the count for a given timepoint reaches 0, that timepoint becomes enabled. To compute the values of ℓ and u, it suffices to maintain two min priority queues (Cormen et al., 2022), one for ℓ and one for u. When a TP X becomes enabled, it is inserted into both queues using its $\mathtt{lb}(X)$ and $\mathtt{ub}(X)$ values as keys. To compute the desired minimum values requires only "peeking" at the current minimum value. TPs need not be extracted from the queues when executed, but instead can be extracted lazily, as follows. Whenever a "peek" reveals a value based on an already-executed TP, that TP can be extracted at that time; and subsequent peek/extractions can be done until a peek reveals a value based on a not-yet-executed TP. In this way, each TP is inserted and extracted exactly once which, together with at most m "decrease key" updates, yields a total cost of $O(m + n\log n)$. The peeks can be done in constant time and so don't affect the overall time. For full flexibility, $O(n)$ worst-case time is required for selecting the timepoint X to execute next, which drives the overall $O(n^2)$ worst-case time. The selection of the time t at which to execute X, if done randomly, can be done in constant time. Of course, an application may have domain-specific criteria that would make the selections of X and t more time-consuming, but that is beyond the purview of the RTE algorithm.

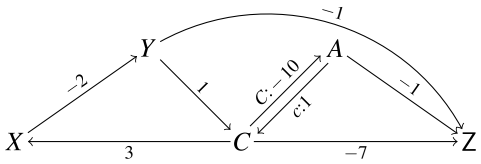


Figure 4: A sample STNU.

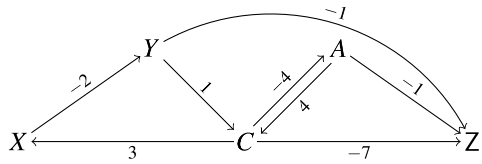


Figure 5: The projection of the sample STNU onto $\omega = (4)$.

3 STNU DISPATCHABILITY

A Simple Temporal Network with Uncertainty (STNU) augments an STN to include *contingent links* that can represent actions with uncertain durations (Morris et al., 2001). An STNU is a triple $(\mathcal{T}, \mathcal{C}, \mathcal{L})$ where $(\mathcal{T}, \mathcal{C})$ is an STN, and $\mathcal{L}$ is a set of contingent links, each of the form (A, x, y, C), where: $A \in \mathcal{T}$ is the *activation timepoint* (ATP); $C \in \mathcal{T}$ is the *contingent timepoint* (CTP); and $0 < x < y < \infty$ specifies bounds on the duration $C - A$. Typically, an executor controls the execution of A, but not C. The execution time for C is only learned in real time, when it happens, but is guaranteed to satisfy $C - A \in [x, y]$. We let $k = |\mathcal{L}|$; and notate the set of contingent timepoints as $\mathcal{T}_c$; and the non-contingent (i.e., executable) timepoints as $\mathcal{T}_x = \mathcal{T} \setminus \mathcal{T}_c$.

Each STNU $(\mathcal{T}, \mathcal{C}, \mathcal{L})$ has a corresponding graph, $(\mathcal{T}, \mathcal{E} \cup \mathcal{E}_{lc} \cup \mathcal{E}_{uc})$, where: $(\mathcal{T}, \mathcal{E})$ is the graph for the STN $(\mathcal{T}, \mathcal{C})$; $\mathcal{E}_{lc}$ is a set of *lower-case* (LC) edges; and $\mathcal{E}_{uc}$ is a set of *upper-case* (UC) edges. The LC and UC edges correspond to the contingent links in $\mathcal{L}$, as follows. For each contingent link $(A, x, y, C) \in \mathcal{L}$, there is an LC edge $A \xrightarrow{c:x} C$ in $\mathcal{E}_{lc}$ and a UC edge $C \xrightarrow{C:-y} A$ in $\mathcal{E}_{uc}$, respectively representing the *uncontrollable possibilities* that the duration $C - A$ might take on its lower bound x or its upper bound y. For convenience, such edges may be notated as $(A, c{:}x, C)$ and $(C, C{:}{-y}, A)$. Figure 4 shows a sample STNU graph with a contingent link $(A, 1, 10, C)$.

An STNU is *dynamically controllable* (DC) if there exists a dynamic strategy for executing its *non-contingent* timepoints such that all of the constraints in $\mathcal{C}$ will necessarily be satisfied no matter how the contingent durations turn out—within their specified bounds (Morris et al., 2001; Hunsberger, 2009). A strategy is dynamic in that it can react in real time to observations of contingent executions, but its execution decisions cannot depend on advance knowledge of contingent durations. As is common in the literature, this paper assumes that strategies can react instantaneously to observations. Morris (2014) presented the first $O(n^3)$-time DC-checking algorithm for STNUs. Cairo et al. (2018) gave a $O(mn + k^2 n + kn \log n)$-time algorithm that is faster on sparse networks.

Most DC-checking algorithms generate a new kind of edge, called a *wait,* that represents a *conditional* constraint. A wait edge $(Y, C{:}{-w}, A)$ represents the conditional constraint that *as long as* C *has not yet executed,* Y must wait until at least w after A. In this paper, a wait labeled by the contingent timepoint C is called a *C-wait*. Following Morris (2014), we define an *extended* STNU (ESTNU) to include a set $\mathcal{C}_w$ of conditional wait constraints, and an ESTNU graph to include a corresponding set $\mathcal{E}_w$ of wait edges. (While wait edges are not necessary for DC-checking, they are typically necessary for dispatchability.)

Morris (2014) defined the dispatchability of an ESTNU in terms of its STN *projections.* A projection of an ESTNU is the STN that results from assigning fixed durations to its contingent links (Morris et al., 2001; Morris, 2014; Hunsberger and Posenato, 2023).

Definition 3 (Projection)**.** *Let $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L}, \mathcal{C}_w)$ be an ESTNU, where $\mathcal{L} = \{(A_i, x_i, y_i, C_i) \mid 1 \leq i \leq k\}$. Let $\omega = (\omega_1, \omega_2, \ldots, \omega_k)$ be any k-tuple such that $x_i \leq \omega_i \leq y_i$ for each i. Then the* projection *of $\mathcal{S}$ onto ω is the STN $\mathcal{S}_\omega = (\mathcal{T}, \mathcal{C} \cup \mathcal{C}^\omega_{lc} \cup \mathcal{C}^\omega_{uc} \cup \mathcal{C}^\omega_w)$ given by:*

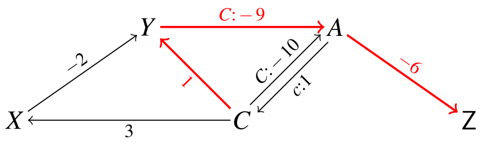
$$
\begin{aligned}
\mathcal{C}^\omega_{lc} &= \{(A_i, \omega_i, C_i) \mid 1 \leq i \leq k\} \\
\mathcal{C}^\omega_{uc} &= \{(C_i, -\omega_i, A_i) \mid 1 \leq i \leq k\} \\
\mathcal{C}^\omega_{w} &= \{(X, -\min\{w, \omega_i\}, A_i) \mid \\
&\qquad\qquad (X, C_i{:}-w, A_i) \in \mathcal{C}_w\}
\end{aligned}
$$

The constraints in $\mathcal{C}^\omega_{lc} \cup \mathcal{C}^\omega_{uc}$ together fix the duration of each contingent link (A_i, x_i, y_i, C_i) to $C_i - A_i = \omega_i$. Each wait edge $(X, C_i{:}{-w}, A_i) \in \mathcal{C}_w$ projects onto *either* the STN edge $(X, -w, A_i)$ if $w \leq \omega_i$ (i.e., if the wait expires before C_i executes) *or* the STN edge $(X, -\omega, A_i)$ (i.e., if C_i executes before $A_i + w$).

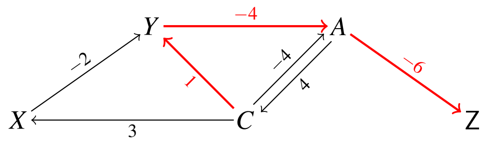
Figure 5 shows the projection of the sample STNU from Figure 4 onto $\omega = (4)$. Note that this projection is *not* dispatchable (as an STN) since, for example, there is no shortest path from C to Y that is a vee-path.

Definition 4 (ESTNU dispatchability (Morris, 2014))**.** *An ESTNU is dispatchable if* all *of its STN projections are dispatchable (as STNs).*

Morris (2014) argued informally that a dispatchable ESTNU (Definition 4) would provide a real-time execution guarantee, but did not specify an RTE algorithm for ESTNUs. However, he showed that



(a) Dispatchable ESTNU.



(b) Its projection onto $\omega = (4)$.

Figure 6: A dispatchable ESTNU (top) that is equivalent to the STNU from Figure 4 and one of its projections (bottom).

Algorithm 2: RTE*: real-time execution for ESTNUs.

Input: $\mathcal{S} = (\mathcal{T}_x \cup \mathcal{T}_c, \mathcal{C}, \mathcal{L}, \mathcal{C}_w)$, an ESTNU
Output: A function $f : (\mathcal{T}_x \cup \mathcal{T}_c) \to \mathbb{R}$ or `fail`

1 D := $\text{RTE}^*_{init}(\mathcal{T}_x, \mathcal{T}_c)$ // Initialization
2 **while** $\mathtt{D}.\mathcal{U}_x \cup \mathtt{D}.\mathcal{U}_c \neq \emptyset$ **do** // Some TPs unexec.
3 $\Delta := \text{RTE}^*_{genD}(\mathtt{D})$ // Generate exec. decision
4 **if** $\Delta = \mathtt{fail}$ **then**
5 **return** `fail`
6 $(\rho, \tau) := \mathtt{Observe}_c(\mathcal{S}, \mathtt{D}, \Delta)$ // Observe CTPs
7 D := $\text{RTE}^*_{update}(\mathtt{D}, \Delta, (\rho, \tau))$ // Update in D
8 **if** D = `fail` **then**
9 **return** `fail`
10 **return** $\mathtt{D}.f$

his $O(n^3)$-time DC-checking algorithm, modified to generate wait edges, outputs an equivalent dispatchable ESTNU when given a DC input. Hunsberger and Posenato (2023) recently provided an $O(mn + kn^2 + n^2 \log n)$-time algorithm that is faster on sparse graphs.

Figure 6(a) shows a dispatchable ESTNU that is equivalent to the STNU from Figure 4. Figure 6(b) shows its projection onto $\omega = (4)$, which *is* dispatchable (as an STN).

4 RTE ALGORITHM FOR ESTNUs

This section specifies a real-time execution algorithm for ESTNUs, called RTE*, whose high-level iterative operation is given as Algorithm 2.

On each iteration, the algorithm first generates an execution decision (Line 3). Next, it *observes* whether any contingent TPs happened to execute (Line 6). Since, as discussed below, the execution of contingent TPs is not controlled by the RTE* algorithm, observation is represented here by an *oracle*, $\mathtt{Observe}_c$. Afterward, the RTE* algorithm responds by updating information (Line 7). In successful instances, the RTE* algorithm returns a complete set of variable assignments for the timepoints in $\mathcal{T}$ (equivalently, a function $f : \mathcal{T} \to \mathbb{R}$).

Algorithm 3: RTE^*_{init}: Initialization.

Input: $\mathcal{T}_x$, executable TPs; $\mathcal{T}_c$, contingent TPs
Output: D, initialized `RTEdata` structure

D = new(`RTEdata`)
$\mathtt{D}.\mathcal{U}_x := \mathcal{T}_x$; $\mathtt{D}.\mathcal{U}_c := \mathcal{T}_c$; D.now = 0; $\mathtt{D}.f = \emptyset$
$\mathtt{D.Enabs}_x = \{X \in \mathcal{T}_x \mid X$ has no outgoing negative edges$\}$
foreach $X \in \mathcal{T}_x$ **do**
$\mathtt{D.TW}(X) := [0, \infty)$
$\mathtt{D.AcWts}(X) := \emptyset$
return D

The RTE* algorithm maintains information in a data structure, called `RTEdata`, that has the following fields:

- $\mathcal{U}_x$ (the unexecuted *executable* timepoints),
- $\mathcal{U}_c$ (the unexecuted *contingent* timepoints),
- $\mathtt{Enabs}_x$ (the enabled *executable* timepoints),
- `now` (the current time),
- f (a set of variable assignments),
- for each executable timepoint $X \in \mathcal{T}_x$, $\mathtt{TW}(X) = [\mathtt{lb}(X), \mathtt{ub}(X)]$ (time window for X),
- $\mathtt{AcWts}(X)$ (the *activated waits* for X, see below).

A new `RTEdata` instance, D, is initialized by the RTE^*_{init} algorithm (Algorithm 3). Note that for ESTNUs, an *executable* timepoint X is enabled if all of its outgoing negative edges—*including wait edges*—point at already executed timepoints.

Activated Waits. A *wait* edge such as $(X, C{:}{-w}, A)$ represents a conditional constraint that *as long as C has not yet executed, X* must wait at least w after A. Once the activation timepoint A for the contingent link (A, x, y, C) has been executed, say, at some time a, we say that the wait edge has been *activated*, which the RTE* algorithm keeps track of by inserting an entry $(a + w, C)$ into the set $\mathtt{AcWts}(X)$. There are two ways for this wait to be satisfied: C can execute early (i.e., before $a + w$) or the wait can expire (i.e., the current time passes $a + w$). In response to either event, the entry $(a + w, C)$ is removed from $\mathtt{AcWts}(X)$. In general, if $\mathtt{AcWts}(X)$ is non-empty, X cannot be executed.

Algorithm 4: RTE$^*_{genD}$: Generate execution decision.

Input: D, an `RTEdata` structure
Output: Exec decn: `Wait` or (t,V); or `fail`

1 **if** $\texttt{D.Enabs}_x = \emptyset$ **then**
2   **return** `Wait`
3 **foreach** $X \in \texttt{D.Enabs}_x$ **do**
  // Maximum wait time for X
4   $\texttt{wt}(X) = \max\{w \mid \exists(w,\_) \in \texttt{D.AcWts}(X)\}$
  // Greatest lower bound for X
5   $\texttt{glb}(X) = \max\{\texttt{D.lb}(X), \texttt{wt}(X)\}$
 // Earliest possible next execution
6 $t_L = \min\{\texttt{glb}(X) \mid X \in \texttt{D.Enabs}_x\}$
 // Latest possible next execution
7 $t_U = \min\{\texttt{D.ub}(X) \mid X \in \texttt{D.Enabs}_x\}$
8 **if** $[t_L, t_U] \cap [\texttt{D.now}, \infty) = \emptyset$ **then**
9   **return** `fail`
10 Select any $V \in \texttt{D.Enabs}_x$ for which $[\texttt{glb}(X), \texttt{ub}(X)] \cap [\texttt{D.now}, t_U] \neq \emptyset$
11 Select any $t \in [\texttt{glb}(V), \texttt{ub}(V)] \cap [\texttt{D.now}, t_U]$
12 **return** (t,V)

Generate Execution Decision. Hunsberger (2009) formally characterized dynamic execution strategies for STNUs in terms of *real-time execution decisions* (RTEDs). An RTED can have one of two forms: `Wait` or (t,χ). A `Wait` decision can be glossed as "wait for a contingent timepoint to execute". A (t,χ) decision can be glossed as "if no contingent timepoints execute before time t, then execute the timepoints in the set χ". Given the assumption about instantaneous reactivity, it suffices to limit χ to a single timepoint.

Algorithm 4 computes the next RTED for one iteration of the RTE* algorithm. First, at Line 1, if there are no enabled timepoints, then the only viable RTED is `Wait`. Otherwise, the algorithm generates an RTED of the form (t,V) for some $t \in \mathbb{R}$ and some enabled TP V. Lines 3 to 5 compute, for each enabled TP X, the maximum wait time $\texttt{wt}(X)$ among all of X's activated waits (or $-\infty$ if there are none), and then compares that with the lower-bound $\texttt{lb}(X)$ from X's time window to generate the earliest time, $\texttt{glb}(X)$, at which X could be executed.[4] Then, at Line 6, it computes the earliest possible time t_L that *any* enabled TP could be executed next. Line 7 computes the latest time at which the next execution event could occur. The algorithm fails if the interval between the earliest possible time and the latest does not include times at or after now (Line 9). Otherwise, it selects any one of the enabled timepoints V whose time window includes times in $[\texttt{D.now}, t_U]$ (Line 10); and any time $t \in [\texttt{glb}(V), \texttt{ub}(V)] \cap [\texttt{D.now}, t_U]$ at which to execute it (Line 11). (Note the flexibility inherent in the selection of both V and t.) The algorithm outputs the RTED (t,V) (Line 12).

[4] $\texttt{D.lb}(X)$ and $\texttt{D.ub}(X)$ respectively denote the lower and upper bounds of X's time window, $\texttt{D.TW}(X)$.

Algorithm 5: $\texttt{Observe}_c$: Oracle.

Input: $\mathcal{S} = (\mathcal{T}_x \cup \mathcal{T}_c, \mathcal{C}, \mathcal{L}, \mathcal{C}_w)$, an ESTNU; D, an `RTEdata` structure; Δ, an RTED
Output: (ρ, τ), where $\rho \in \mathbb{R}$ and $\tau \subseteq \texttt{D}.\mathcal{U}_c$

1 $f := \texttt{D}.f$; $\texttt{now} := \texttt{D.now}$
 // Get $ACLs$: currently active contingent links
2 $ACLs := \{(A,x,y,C) \in \mathcal{L} \mid f(A) \leq \texttt{now}, f(C) = \bot\}$
 // Waiting forever
3 **if** $ACLs = \emptyset$ **and** $\Delta = \texttt{Wait}$ **then**
4   **return** $(\infty, \emptyset)$
 // No CTPs execute at or before time t
5 **if** $ACLs = \emptyset$ **and** $\Delta = (t,V)$ **then**
6   **return** $(t, \emptyset)$
 // Compute bounds for possible contingent executions
7 $lb_c := \min\{f(A) + x \mid (A,x,y,C) \in ACLs\}$
8 $ub_c := \min\{f(A) + y \mid (A,x,y,C) \in ACLs\}$
9 Select any $t_c \in [lb_c, ub_c]$
 // Oracle decides not to execute any CTPs yet
10 **if** $\Delta = (t,V)$ **and** $t_c > t$ **then**
11   **return** $(t, \emptyset)$
 // Oracle decides to execute one or more CTPs
12 $\tau^* := \{C \mid (A,x,y,C) \in ACLs, t_c \in [a+x, a+y]$, where $a = f(A)\}$
13 Select τ: any *non-empty* subset of τ^*
14 **return** (t_c, τ)

Observation. Once the RTE* algorithm generates an execution decision (e.g., "If nothing happens before time t, then execute V"), it must wait to see what happens (e.g., whether some contingent timepoints happen to execute). Since the execution of contingent TPs is not controlled by the RTE* algorithm, we represent it within the algorithm by an *oracle,* called $\texttt{Observe}_c$, whose pseudocode is given in Algorithm 5.

The oracle, $\texttt{Observe}_c$, non-deterministically decides whether to execute any contingent TPs and, if so, when. At Line 2, it computes the set of *currently active* contingent links (i.e., those whose activation TPs have been executed, but whose contingent TPs have not yet). If there are none, then no CTPs can execute. In that case, $\texttt{Observe}_c$ returns $(\infty, \emptyset)$ in response to a `wait` decision (Line 4), or $(t, \emptyset)$ in response to a (t,V) decision (Line 6). Otherwise (i.e., there are some active contingent links), $\texttt{Observe}_c$ computes the range of possible times for the next contingent

Algorithm 6: RTE^*_{update}: update information in $\texttt{D}$.

Input: $\mathcal{S}$, an ESTNU; $\texttt{D}$, an $\texttt{RTEdata}$ structure; Δ, an RTED ($\texttt{Wait}$ or (t,V)); (ρ,τ), an observation, where $\rho \in \mathbb{R}$ and $\tau \subseteq \texttt{D}.\mathcal{U}_c$

Output: Updated $\texttt{D}$ or $\texttt{fail}$

// Case 0: Failure (waiting forever)

1 **if** $\rho = \infty$ **then**

2   **return** $\texttt{fail}$

// Case 1: Only contingent timepoints executed

3 **if** $\Delta = \texttt{Wait}$ **or** $(\Delta = (t,V)$ **and** $\rho < t)$ **then**

4   **HCE**$(\mathcal{S},\texttt{D},\rho,\tau)$

5 **else**

    // Case 2: Executable timepoint V executes at t

6   **HXE**$(\mathcal{S},\texttt{D},t,V)$

    // Case 3: CTPs **also** execute at t

7   **if** $\tau \neq \emptyset$ **then**

8   **HCE**$(\mathcal{S},\texttt{D},t,\tau)$

9 $\texttt{D.now} := \rho$

10 **return** $\texttt{D}$

Algorithm 7: **HCE**: Handle contingent executions.

Input: $\mathcal{S}$, an ESTNU; $\texttt{D}$, an $\texttt{RTEdata}$; $\rho \in \mathbb{R}$, an execution time; $\tau \subseteq \mathcal{U}_c$, CTPs to execute at ρ

Result: $\texttt{D}$ updated

1 **foreach** $C \in \tau$ **do**

2   Add (C,ρ) to $\texttt{D}.f$

3   Remove C from $\texttt{D}.\mathcal{U}_c$

4   Update time windows for neighbors of C

5   Remove C-waits from all $\texttt{D.AcWts}$ sets

6   Update $\texttt{D.Enabs}_x$ due to incoming neg. edges to C or any deleted C-waits

Algorithm 8: **HXE**: Handle a non-contingent execution.

Input: $\mathcal{S}$, an ESTNU; $\texttt{D}$, an $\texttt{RTEdata}$ structure; $t \in \mathbb{R}$; $V \in \mathcal{U}_x$

Result: $\texttt{D}$ updated

1 Add (V,t) to $\texttt{D}.f$

2 Remove V from $\texttt{D}.\mathcal{U}_x$

3 Update time windows for neighbors of V

4 Update $\texttt{D.Enabs}_x$ due to any negative incoming edges to V

5 **if** *V is activation TP for some CTP C* **then**

6   **foreach** $(Y, C{:}{-w}, V) \in \mathcal{E}_w$ **do**

7   Insert $(t+w, C)$ into $\texttt{D.AcWts}(Y)$

execution event and arbitrarily selects some time t_c within that range (Lines 7 to 9). Now, if the pending RTE* decision is (t,V), and t_c happens to be greater than t, then the oracle has effectively decided not to execute any contingent TPs yet (Line 10). Otherwise, it computes the set τ^* of CTPs that *could* execute at time t_c (Line 12) and then arbitrarily selects a non-empty subset of τ^* to *actually* execute at time t_c (Line 13).

Update. The response of the RTE* algorithm to its observation of possible CTP executions is handled by the RTE^*_{update} algorithm (Algorithm 6). If $\rho = \infty$, which can only happen when a $\texttt{Wait}$ decision was made but there were no active contingent links, then the RTE* algorithm would wait forever and, hence, fail (Line 2). Otherwise, $\rho < \infty$. If the decision was $\texttt{wait}$, then one or more contingent TPs must have executed at ρ (and no executable TPs), whence (Lines 3 to 4) the relevant updates are computed by the **HCE** algorithm (Algorithm 7). The same updates are also needed if the decision was (t,V), where $\rho < t$ (Lines 3 to 4).

The **HCE** algorithm (Algorithm 7) updates $\texttt{D}$ in response to contingent executions as follows. Lines 2 to 3 record that C occurred at ρ by adding the variable assignment (C,ρ) to $\texttt{D}.f$ and removing C from $\texttt{D}.\mathcal{U}_c$. Line 4 updates the time windows for neighboring timepoints, exactly like the RTE algorithm for STNs. Since the execution of C automatically satisfies all C-waits, Line 5 removes any C-waits from the $\texttt{D.AcWts}$ sets. Finally, Line 6 updates the set of enabled executable TPs in case the execution of C or the deletion of C-waits enables some new TPs.

In the remaining cases (Lines 5 to 8) of RTE^*_{update} (Algorithm 6), the decision is (t,V) and $\rho = t$. In other words, no contingent TPs executed *before* time t and, so, the executable timepoint V must be executed at t. The corresponding updates are handled by the **HXE** algorithm (Algorithm 8). The **HXE** updates are the same as those done by the RTE algorithm for STNs, except that if V happens to be an activation TP for some contingent TP C, then information about all C-waits must be entered into the appropriate $\texttt{AcWts}$ sets (Lines 5 to 7).

Finally, in the (extremely rare) case (of Algorithm 6, Line 8) where one or more CTPs happen to execute precisely at time t (i.e., simultaneously with V), the **HCE** algorithm (Algorithm 7) performs the needed updates, as in Case 1. Finally, Algorithm 6 updates the current time to ρ (Line 9).

Table 2 shows sample runs of the RTE* algorithm on the *dispatchable* ESTNU from Figure 6(a). In Table 2(a), C executes early (at $A+5$); in Table 2(b), C executes late (at $A+10$). Both runs result in variable assignments that satisfy all of the constraints in $\mathcal{C}$.

Table 2: Sample runs of the RTE* algorithm on the *dispatchable* ESTNU from Figure 6(a).
(a) Sample run where C executes early (at $A+5$).

Iter.	$\texttt{TW}(A)$	$\texttt{TW}(X)$	$\texttt{TW}(Y)$	$\texttt{AcWts}(A)$	$\texttt{AcWts}(X)$	$\texttt{AcWts}(Y)$	now	$\texttt{Enabs}_x$	RTED	Obs	Exec
Init.	$[0,\infty)$	$[0,\infty)$	$[0,\infty)$	$\emptyset$	$\emptyset$	$\emptyset$	0	$\{Z\}$	$(0,Z)$	$(0,\emptyset)$	$Z := 0$
1	$[6,\infty)$	$[0,\infty)$	$[0,\infty)$	$\emptyset$	$\emptyset$	$\emptyset$	0	$\{A\}$	$(7,A)$	$(7,\emptyset)$	$A := 7$
2	—	$[0,\infty)$	$[0,\infty)$	—	$\emptyset$	$\{(16,C)\}$	7	$\{Y\}$	$(16,Y)$	$(12,\{C\})$	$C := 12$
3	—	$[0,15]$	$[0,13]$	—	$\emptyset$	$\emptyset$	12	$\{Y\}$	$(13,Y)$	$(13,\emptyset)$	$Y := 13$
4	—	$[15,15]$	—	—	$\emptyset$	—	13	$\{X\}$	$(15,X)$	$(15,\emptyset)$	$X := 15$

(b) Sample run where C executes late (at $A+10$).

Iter.	$\texttt{TW}(A)$	$\texttt{TW}(X)$	$\texttt{TW}(Y)$	$\texttt{AcWts}(A)$	$\texttt{AcWts}(X)$	$\texttt{AcWts}(Y)$	now	$\texttt{Enabs}_x$	RTED	Obs	Exec
Init.	$[0,\infty)$	$[0,\infty)$	$[0,\infty)$	$\emptyset$	$\emptyset$	$\emptyset$	0	$\{Z\}$	$(0,Z)$	$(0,\emptyset)$	$Z := 0$
1	$[6,\infty)$	$[0,\infty)$	$[0,\infty)$	$\emptyset$	$\emptyset$	$\emptyset$	0	$\{A\}$	$(7,A)$	$(7,\emptyset)$	$A := 7$
2	—	$[0,\infty)$	$[0,\infty)$	—	$\emptyset$	$\{(16,C)\}$	7	$\{Y\}$	$(16,Y)$	$(16,\emptyset)$	$Y := 16$
3	—	$[18,\infty)$	—	—	$\emptyset$	—	16	$\{X\}$	$(22,X)$	$(17,\{C\})$	$C := 17$
4	—	$[18,20]$	—	—	$\emptyset$	—	17	$\{X\}$	$(19,X)$	$(19,\emptyset)$	$X := 19$

RTE* Complexity. The worst-case complexity of the RTE* algorithm is similar to that of the RTE algorithm except for the maintenance of the `AcWts` sets (which is handled by the **HCE** and **HXE** algorithms). The `AcWts` sets can also be implemented using min priority queues. Since there are at most nk wait edges, each of which gets inserted into an `AcWts` set exactly once, and also gets deleted exactly once, the worst-case complexity over the entire RTE* algorithm is $O(nk + (nk)\log(nk)) = O(nk\log(nk))$. This assumes that the deletions are done lazily, as described earlier for the other min priority queues. Therefore, the overall complexity is $O(m+n\log n+nk\log(nk)) = O(m+nk\log(nk))$. Finally, although we provide pseudocode for the $\texttt{Observe}_c$ oracle, that was just to highlight the range of possible observations. From the perspective of the RTE* algorithm, the oracle presents observations in real time and, hence, there is no computation cost associated with them.

4.1 Main Theorem

Theorem 2. *Let $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L}, \mathcal{C}_w)$ be an ESTNU. Every run of the RTE* algorithm on $\mathcal{S}$ corresponds to a run of the RTE algorithm for STNs on some STN projection $\mathcal{S}_\omega$ of $\mathcal{S}$, yielding the same variable assignments to the timepoints in $\mathcal{T}$.*

The following definitions, closely related to definitions in Morris (2016) and Hunsberger (2009), are used in the proof.

Definition 5 (Execution sequence)**.** *A (possibly partial)* execution sequence *is any sequence of the form $\sigma = ((X_1,t_1),(X_2,t_2),\ldots,(X_h,t_h))$ where $\{X_1,X_2,\ldots,X_h\} \subseteq \mathcal{T}$ and $t_1 \leq t_2 \leq \ldots \leq t_h$. For any $(X,t) \in \sigma$, we write $\sigma(X) = t$. For any X that doesn't appear in σ, we write $\sigma(X) = \perp$. In addition, we let $\max(\sigma) = t_h$ notate the time of the latest execution event in σ.*

Note that the "functions" $\texttt{D}.f$ and f that are incrementally computed by the RTE* and RTE algorithms may be viewed as execution sequences; and that $\texttt{D.now} = \max(\texttt{D}.f)$ and $\texttt{now} = \max(f)$.

Definition 6 (Pre-history)**.** *The* pre-history π_σ *of an execution sequence $\sigma = ((X_1,t_1),\ldots,(X_h,t_h))$ is a set that specifies the duration, $\sigma(C)-\sigma(A)$, of each contingent link (A,x,y,C) for which $\sigma(A),\sigma(C) \leq \max(\sigma)$, and constrains the duration of any* currently active *contingent link (A',x',y',C'), where $\sigma(A') \leq \max(\sigma)$ but $\sigma(C') = \perp$, to $C'-A' \geq \max(\sigma) - A'$ (i.e., $C' \geq \max(\sigma)$).*

Definition 7 (Respect)**.** *A projection $\mathcal{S}_\omega$* respects *a pre-history π if it is consistent with the constraints on the durations specified by π.*

Definition 8 (RTE-compliant)**.** *A (possibly partial) execution sequence σ is* RTE-compliant *for an ESTNU $\mathcal{S}$ if it can be generated by some run of the RTE algorithm on* every *projection $\mathcal{S}_\omega$ that respects the pre-history π_σ.*

Proof. This proof incrementally analyzes an arbitrary execution sequence generated by the RTE* algorithm on the ESTNU $\mathcal{S}$, placing no restrictions on the choices it makes along the way, while constructing in parallel a corresponding run of the RTE algorithm on an incrementally specified projection of $\mathcal{S}$ such that, in the end, both algorithms generate the same set of variable assignments. In what follows, information computed by RTE* is prefixed by `D`; non-prefixed terms by RTE. The proof uses induction to show that at the beginning of each iteration the following invariants hold:

(P1) $\mathtt{D}.f = f$ (i.e., the current, typically partial execution sequences are the same); and

(P2) f is RTE compliant for $\mathcal{S}$.

Base Case. $\mathtt{D}.f = \emptyset = f$, and $\emptyset$ is trivially RTE-compliant for $\mathcal{S}$.

Recursive Case. Suppose (P1) and (P2) hold at the beginning of some iteration. First, note that $\mathtt{D}.f = f$ implies that $\mathtt{D}.\mathcal{U}_x \cup \mathtt{D}.\mathcal{U}_c = \mathcal{U}$. In the case where these sets are both empty, both algorithms terminate, signaling that $\mathtt{D}.f = f$ is a complete assignment. Otherwise, both sets are non-empty and we must show that (P1) and (P2) hold at the start of the *next* iteration.

Note that $\mathtt{D.now} = \max(\mathtt{D}.f) = \max(f) = \mathtt{now}$. Next, we show that $\mathtt{D.Enabs}_x = \mathtt{Enabs} \cap \mathcal{U}_x$. This follows because each negative edge in $\mathcal{S}$ is either an ordinary edge or a wait edge, both of which project onto negative edges in every projection. Since $\mathtt{D.Enabs}_x$ only includes executable TPs, the equality holds.

Case 1: $\mathtt{D.Enabs}_x = \emptyset$. Therefore, $\mathtt{Enabs} \subseteq \mathcal{U}_c$. Then the RTE* algorithm generates a $\mathtt{Wait}$ decision. Now, $\mathtt{Enabs} = \emptyset$ would cause RTE to fail (Algorithm 1, Line 7), contradicting the dispatchability of any STN projection from this point onward. Therefore, $\mathtt{Enabs} \neq \emptyset$ and, thus, there exists at least one enabled CTP C which, given the negative edge from C to its activation TP, implies that its contingent link is currently active. Therefore, Lines 7 to 13 of the oracle (Algorithm 5) would select an observation of the form (t_c, τ), where $\tau \neq \emptyset$.

Now, by (P2), f is RTE-compliant; hence it can be generated by any projection that respects the pre-history π_f. Next, let f' be the execution sequence obtained by executing the CTPs in τ at time t_c; and let $\pi_{f'}$ be the corresponding pre-history. Among the projections that respect the pre-history π_f are those that also respect $\pi_{f'}$. Since the RTE algorithm, when applied to any of those projections, *must* execute the CTPs in τ at time t_c, it follows that f' is RTE compliant for $\mathcal{S}$ (i.e., (P2) holds at the start of the next iteration). And since the **HCE** algorithm executes the CTPs in τ at t_c, it follows that (P1) holds at the start of the next iteration. Finally, the other updates done by **HCE** are equivalent to those done by RTE, as follows. Removing any C-waits for $C \in \tau$ corresponds to the satisfaction of the corresponding projected constraints since, for example, a C-wait $(W, C{:}{-8}, A)$ projects to the negative edge $(W, -5, A)$ in the projection where $C - A = 5$, whose lower bound of $A + 5$ is automatically satisfied when C executes at $A + 5$. And RTE*'s updating of $\mathtt{D.Enabs}_x$ is equivalent to RTE's updating of $\mathtt{Enabs}$ given that wait edges project onto ordinary negative edges.

Case 2: $\mathtt{D.Enabs}_x \neq \emptyset$. Here, the RTE$^*_{genD}$ algorithm (Algorithm 4) would, at Lines 3 to 12, generate an execution decision of the form (t, V). Now, for any (executable) $X \in \mathtt{Enabs}_x$, its upper bound is computed based solely on propagations from executed TPs along non-negative edges. Given that $\mathtt{D}.f = f$, it follows that $\mathtt{D.ub}(X) = \mathtt{ub}(X)$ for each $X \in \mathtt{D.Enabs}_x$, regardless of the f-respecting projection that RTE is applied to. Similar remarks apply to the lower bound for each X except that $\mathtt{D.glb}(X) \geq \mathtt{lb}(X)$. To see this, note that although propagations along ordinary negative edges done by RTE* are identical to those done by RTE, the activated waits in $\mathtt{AcWts}(X)$ may impose stronger constraints. For example, consider an activated wait edge $(X, C_i{:}{-7}, A_i)$, which imposes a lower bound of $A_i + 7$ on X. In a projection where $\omega_i = 4 < 7$, this edge projects onto the ordinary negative edge $(X, -4, A_i)$, which imposes the weaker lower bound of $A_i + 4$ on X. In contrast, in a projection where $\omega_i = 9 \geq 7$, the wait edge projects onto the ordinary negative edge $(X, -7, A_i)$, which imposes the lower bound of $A_i + 7$ on X. In general, it therefore follows that $[\mathtt{D}.glb(X), \mathtt{D}.ub(X)] \subseteq [\mathtt{lb}(X), \mathtt{ub}(X)]$. To ensure that the RTE$^*_{genD}$ algorithm does not fail at Line 9, we must show that $\mathtt{D.glb}(X) \leq \mathtt{D.ub}(X)$. To see why, let $\mathcal{S}_\omega$ be any projection that respects f, but also specifies maximum durations for all of the currently active contingent links. In $\mathcal{S}_\omega$, all C-waits project onto negative edges of the same length, which implies that $\mathtt{D.glb}(X) = \mathtt{lb}(X) \leq \mathtt{ub}(X) = \mathtt{D.ub}(X)$, since the dispatchability of all projections ensures that RTE cannot fail, and hence $[\mathtt{D}.tl, \mathtt{D}.ul] \cap [\mathtt{D.now}, \infty) \neq \emptyset$. Therefore, RTE$^*_{genD}$ will generate an RTED of the form (t, V).

Case 2a: A (ρ, τ) *observation, where* $\rho < t$. This case can be handled similarly to Case 1.

Case 2b: A $(t, \emptyset)$ *observation..* Here, RTE* executes V at time t. Since $t \in [\mathtt{D.glb}(V), \mathtt{D.ub}(V)] \subseteq [\mathtt{lb}(V), \mathtt{ub}(V)]$ it follows that executing V at t is a viable choice for the RTE algorithm for *every* projection that (1) respects f; and (2) constrains the duration of each active contingent link (A, x, y, C) to satisfy $C - A \geq t - f(A)$. (And such projections exist, since otherwise the *oracle* could not have generated the observation $(t, \emptyset)$.) Therefore, (P1) and (P2) will necessarily hold at the start of the next iteration, when RTE is restricted to such projections. Finally, note that the updates done by the **HXE** algorithm are exactly the same as those done by RTE, except for the updating of the activated waits in the case where V happens to be an activation timepoint. However, inserting an entry $(t + w, C_i)$ into the set $\mathtt{D.AcWts}(Y)$ in response to a wait edge $(Y, C_i{:}{-w}, V)$, merely ensures that the bound for the corresponding projected edge $(Y, -\gamma, V)$ will be respected by RTE*, where $\gamma = \min\{w, \omega_i\}$ and ω_i specifies the duration of the relevant contingent link.

Case 2c: A (t,τ) *observation, where* $\tau \neq \emptyset$. This case is similar to a combination of Case 1 (with $\rho = t$) and Case 2b. □

Corrolary 1. *An ESTNU* $\mathcal{S}$ *is dispatchable if and only if every run of the RTE** *algorithm on* $\mathcal{S}$ *outputs a solution for the ordinary constraints in* $\mathcal{S}$.

Proof. By Theorem 2, $\mathcal{S}$ is dispatchable if and only if each run of RTE* generates a complete assignment that can also be generated by a run of RTE on some projection $\mathcal{S}_\omega$. But by Definitions 4 and 1, $\mathcal{S}$ is dispatchable if and only if every one of its STN projections is dispatchable (i.e., every run of RTE on any of the STN projections generates a solution). □

5 CONCLUSION

The main contributions of this paper are:

1. to provide a formal definition of a real-time execution algorithm for ESTNUs, called RTE*, that provides maximum flexibility while requiring only minimal computation; and
2. to formally prove that an ESTNU $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L}, \mathcal{C}_w)$ is dispatchable (according to the definition in the literature) if and only if every run of RTE* on $\mathcal{S}$ necessarily satisfies all of the constraints in $\mathcal{C}$ no matter how the contingent durations play out in real time.

In so doing, the paper fills an important gap in the algorithmic and theoretic foundations of the dispatchability of Simple Temporal Networks with Uncertainty.

Since the worst-case complexity of the RTE* algorithm is $O(m + nk\log(nk))$, future work will focus on generating equivalent dispatchable ESTNUs having the minimum number of (ordinary and wait) edges.

REFERENCES

Cairo, M., Hunsberger, L., and Rizzi, R. (2018). Faster Dynamic Controllablity Checking for Simple Temporal Networks with Uncertainty. In *25th International Symposium on Temporal Representation and Reasoning (TIME-2018)*, volume 120 of *LIPIcs*, pages 8:1–8:16.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022). *Introduction to Algorithms, 4th Edition*. MIT Press.

Dechter, R., Meiri, I., and Pearl, J. (1991). Temporal Constraint Networks. *Artificial Intelligence*, 49(1-3):61–95.

Hunsberger, L. (2009). Fixing the semantics for dynamic controllability and providing a more practical characterization of dynamic execution strategies. In *16th International Symposium on Temporal Representation and Reasoning (TIME-2009)*, pages 155–162.

Hunsberger, L. and Posenato, R. (2023). A Faster Algorithm for Converting Simple Temporal Networks with Uncertainty into Dispatchable Form. *Information and Computation*, 293(105063):1–21.

Morris, P. (2014). Dynamic controllability and dispatchability relationships. In *CPAIOR 2014*, volume 8451 of *LNCS*, pages 464–479. Springer.

Morris, P. (2016). The Mathematics of Dispatchability Revisited. In *26th International Conference on Automated Planning and Scheduling (ICAPS-2016)*, pages 244–252.

Morris, P., Muscettola, N., and Vidal, T. (2001). Dynamic control of plans with temporal uncertainty. In *IJCAI 2001: Proc. of the 17th international joint conference on Artificial intelligence*, volume 1, pages 494–499.

Muscettola, N., Morris, P. H., and Tsamardinos, I. (1998). Reformulating Temporal Plans for Efficient Execution. In *6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR-1998)*, pages 444–452.

Tsamardinos, I., Muscettola, N., and Morris, P. (1998). Fast Transformation of Temporal Plans for Efficient Execution. In *15th National Conf. on Artificial Intelligence (AAAI-1998)*, pages 254–261.