Foundations of Dispatchability for Simple Temporal Networks with Uncertainty

Luke Hunsberger1 and Roberto Posenato2

1Computer Science Department, Vassar College, Poughkeepsie, NY, U.S.A.
2Dipartimento di Informatica, Università degli 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 re-planning. 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 (Muscellotta 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 (Muscellotta et al., 1998). Such networks are called dispatchable. They provide applications with both flexibility and compu-
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\textsuperscript{*}, 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\textsuperscript{*} 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\textsuperscript{*}, and proves its correctness. Section 5 summarizes the contributions of the paper and sketches possible future work.

2 STN DISPATCHABILITY

A Simple Temporal Network (STN) is a pair, (\(\mathcal{T}, C\)), where \(\mathcal{T}\) is a set of real-valued variables called timepoints (TPs) and \(C\) 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 \(m = |C|\). 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.\textsuperscript{1} Each STN has a corresponding graph, (\(\mathcal{T}, E\)), where the timepoints in \(\mathcal{T}\) serve as nodes and each constraint \(Y - X \leq \delta\) in \(C\) corresponds to a labeled directed edge \(X \rightarrow Y\) in \(E\), called an ordinary edge. For convenience, such edges will be notated as \((X, \delta, Y)\).

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-

\textsuperscript{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: (T, C), an STN with graph (T, E)
Output: A function, f : T → [0, ∞) or fail
1 foreach X ∈ T do
2    TW(X) = [0, ∞)
3    U := T; now := 0
4    Enabs := {X ∈ T | X has no outgoing negative edges}
5 while U ≠ {} do
6      if Enabs = ∅ then
7         return fail
8      ℓ := min{lb(W) | W ∈ Enabs}
9      u := min{ub(W) | W ∈ Enabs}
10     if [ℓ, u] ∩ [now, ∞] = ∅ then
11        return fail
12 Select any X ∈ Enabs | TW(X) ∩ [now, u] ≠ ∅
13 Select any t ∈ TW(X) ∩ [now, u]
14 Remove X from U
15 f(X) := t; now := t
16 Propagate f(X) = t to X’s neighbors in E
17 Enabs := {Y ∈ U | all negative edges from Y terminate at TP's not in U}
18 return f
```

In Algorithm 1, lb(X) and ub(X) respectively denote the lower and upper bounds from X’s time window, TW(X).

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 S = (T, C) is dispatchable if every run of the RTE algorithm (Algorithm 1) on the corresponding STN graph G = (T, E) necessarily generates a solution for S.

Muscettola et al. (1998) showed that for consistent STNs, the all-pairs, shortest-paths (APSP) graph is necessarily dispatchable, but its O(n^3) edges cancel the benefits of local propagation. Their O(n^2)-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.
Table 1: Sample runs of the RTE algorithm.

(a) A sample run of the RTE algorithm on the consistent STN from Figure 1.

<table>
<thead>
<tr>
<th>Iter.</th>
<th>Enabs</th>
<th>Tw(Z)</th>
<th>Tw(A)</th>
<th>Tw(C)</th>
<th>Tw(X)</th>
<th>Tw(Y)</th>
<th>l, u</th>
<th>now</th>
<th>Exec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init.</td>
<td>{Z}</td>
<td>[0,∞)</td>
<td>[0,∞)</td>
<td>[0,∞)</td>
<td>[0,∞)</td>
<td>[0,∞)</td>
<td>0,∞</td>
<td>0</td>
<td>Z := 0</td>
</tr>
<tr>
<td>1</td>
<td>{A,Y}</td>
<td>[1,∞)</td>
<td>[7,∞)</td>
<td>[0,∞)</td>
<td>[0,∞)</td>
<td>[0,∞)</td>
<td>0</td>
<td>4</td>
<td>A := 8</td>
</tr>
<tr>
<td>2</td>
<td>{A,X}</td>
<td>[1,∞)</td>
<td>[7.5]</td>
<td>[6,∞)</td>
<td>—</td>
<td>[1,∞)</td>
<td>0</td>
<td>4</td>
<td>8 fail</td>
</tr>
<tr>
<td>3</td>
<td>{C,X}</td>
<td>—</td>
<td>[9.5]</td>
<td>[6,∞)</td>
<td>—</td>
<td>[6.5]</td>
<td>8</td>
<td>8</td>
<td>X := 18</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Iter.</th>
<th>Enabs</th>
<th>Tw(Z)</th>
<th>Tw(A)</th>
<th>Tw(C)</th>
<th>Tw(X)</th>
<th>Tw(Y)</th>
<th>l, u</th>
<th>now</th>
<th>Exec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init.</td>
<td>{Z}</td>
<td>[0,∞)</td>
<td>[0,∞)</td>
<td>[0,∞)</td>
<td>[0,∞)</td>
<td>[0,∞)</td>
<td>0,∞</td>
<td>0</td>
<td>Z := 0</td>
</tr>
<tr>
<td>1</td>
<td>{A,Y}</td>
<td>[1,∞)</td>
<td>[7,∞)</td>
<td>[0,∞)</td>
<td>[0,∞)</td>
<td>[0,∞)</td>
<td>0</td>
<td>4</td>
<td>A := 8</td>
</tr>
<tr>
<td>2</td>
<td>{C,Y}</td>
<td>—</td>
<td>[9,18]</td>
<td>[0,∞)</td>
<td>[0,∞)</td>
<td>[8,∞)</td>
<td>8</td>
<td>8</td>
<td>C := 15</td>
</tr>
<tr>
<td>3</td>
<td>{Y}</td>
<td>—</td>
<td>—</td>
<td>[0,18]</td>
<td>[0,18]</td>
<td>[8,16]</td>
<td>8</td>
<td>16</td>
<td>Y := 16</td>
</tr>
<tr>
<td>4</td>
<td>{X}</td>
<td>—</td>
<td>—</td>
<td>[18,18]</td>
<td>[18,18]</td>
<td>[18,18]</td>
<td>16</td>
<td>18</td>
<td>X := 18</td>
</tr>
</tbody>
</table>

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

Figure 3: An equivalent dispatchable STN graph.

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 $l(X)$ and $ub(X)$ 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/ex 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.
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 \((T, C, L)\) where \((T, C)\) is an STN, and \(L\) is a set of contingent links, each of the form \((A, x, y, C)\), where: \(A \in T\) is the activation timepoint (ATP); \(C \in T\) 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 \(k = |L|\); and notate the set of contingent timepoints as \(T_c\); and the non-contingent (i.e., executable) timepoints as \(T_e = T \setminus T_c\).

Each STNU \((T, C, L)\) has a corresponding graph, \((T, E \cup E_c \cup E_w)\), where: \((T, E)\) is the graph for the STN \((T, C)\); \(E_c\) is a set of lower-case (LC) edges; and \(E_w\) is a set of upper-case (UC) edges. The LC and UC edges correspond to the contingent links in \(L\), as follows. For each contingent link \((A, x, y, C) \in L\), there is an LC edge \(A \xrightarrow{x-y} C\) in \(E_c\) and a UC edge \(C \xrightarrow{x-y} A\) in \(E_w\), 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 \(C\) 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 \(O(n^2)\)-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.

Figure 4: A sample STNU.

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 \(C_w\) of conditional wait constraints, and an ESTNU graph to include a corresponding set \(E_w\) 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 \(S = (T, C, L, C_w)\) be an ESTNU, where \(L = \{(A, i, j, C) \mid 1 \leq i \leq k\}\). 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 \(S\) onto \(\omega\) is the STN \(S_\omega = (T, C, L, C_w')\) given by:

\[
C_{w'}^\omega = \{(A, \omega_i, C) \mid 1 \leq i \leq k\} \\
C_{w_c}^\omega = \{(C, -\omega_i, A_i) \mid 1 \leq i \leq k\} \\
C_{w_u}^\omega = \{(X, -\min\{w_i, \omega_i\}, A_i) \mid (X, C, A_i - w_i, A_i) \in C_w\}
\]

The constraints in \(C_{w_c}^\omega \cup C_{w_u}^\omega\) together fix the duration of each contingent link \((A_i, x_i, y_i, C)\) to \(C_i - A_i = \omega_i\). Each wait edge \((X, C, w, A_i) \in E_w\) 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 \((X, -\omega_i, A_i)\) (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 \(T\) 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
his \(O(n^3)\)-time DC-checking algorithm, modified to
generate wait edges, outputs an equivalent dispatch-
able ESTNU when given a DC input. Hunsberger
and Posenato (2023) recently provided an \(O(mn +
k^2 + n^3 \log n)\)-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 dispatch-
able (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, \(\text{observe}\). After-
ward, the RTE\(^*\) algorithm responds by updating in-
formation (Line 7). In successful instances, the RTE\(^*\)
algorithm returns a complete set of variable assignments
for the timepoints in \(T\) (equivalently, a function
\(f : T \rightarrow \mathbb{R}\)).

The RTE\(^*\) algorithm maintains information in a
data structure, called RTE\(\text{data}\), that has the following
fields:

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

A new RTE\(\text{data}\) instance, \(D\), is initialized by the
RTE\(^{\text{init}}\) algorithm (Algorithm 3). Note that for ES-
TNUs, 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 en-
try \((a + w, C)\) into the set \(\text{AcWts}(X)\). 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 \(\text{AcWts}(X)\). In
general, if \(\text{AcWts}(X)\) is non-empty, \(X\) cannot be executed.

Algorithm 3: RTE\(^{\text{init}}\): Initialization.

\begin{algorithm}
\caption{RTE\(^{\text{init}}\): Initialization.}
\begin{algorithmic}[1]
\Input \(\mathcal{T}_c\), executable TPs; \(\mathcal{T}_i\), contingent TPs
\Output \(D\), initialized RTE\(\text{data}\) structure
\State \(D := \text{new}(\text{RTEdata})\)
\State \(\mathcal{U}_c := \mathcal{T}_i\); \(\mathcal{U}_c := \mathcal{T}_c\); \(\text{now} = 0\); \(D.f = \emptyset\)
\State \(\text{Enabs} = \{X \in \mathcal{T}_c \mid X \text{ has no outgoing negative edges}\}\)
\ForEach \(X \in \mathcal{T}_c\)
\State \(\text{D.Tw}(X) := [0, \infty)\)
\State \(\text{D.AcWts}(X) := \emptyset\)
\EndFor
\State \Return \(D\)
\end{algorithmic}
\end{algorithm}
Algorithm 4: RTE∗

genD: Generate execution decision.
Input: D, an RTE data structure
Output: Exec decn: Wait or (t,V );... (i.e., there are
some active contingent links), Observe computes
the range of possible times for the next contingent
execution of the RTE∗

Algorithm 5: Observe.: Oracle.

Input: S = (τT, τL, C, L, Cc), an ESTNU; D, an
RTE data structure; Δ, an RTED
Output: (p,τ), where p ∈ R and τ ⊆ D.UL

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 exe-
cute 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 it-
eration 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 ∈ R and some
enabled TP V. Lines 3 to 5 compute, for each enabled
TP X, the maximum wait time X mái among all of
X’s activated waits (or −∞ if there are none), and then
computes that with the lower-bound lb(x) from X’s
time window to generate the earliest time, glb(X), at
which X could be executed.4 Then, at Line 6, it com-
putes the earliest possible time tL 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 in-
cludes times in [D.now,tf] (Line 10); and any time

4D.lb(X) and D.ub(X) respectively denote the lower and
upper bounds of X’s time window, D.TW(X).

Observation. Once the RTE∗ algorithm generates
an execution decision (e.g., “If nothing happens be-
fore time t, then execute V”), it must wait to see
what happens (e.g., whether some contingent time-
points happen to execute). Since the execution of
contingent TPs is not controlled by the RTE∗ algo-

4}
Algorithm 6: RTE\textsuperscript{update} - update information in D.

\textbf{Input:} S, an ESTNU; D, an RTE\textsuperscript{data} structure; 
\Delta, an RTE\textsuperscript{D} (Wait or (t, V)); (p, \tau), an 
observation, where \( p \in \mathbb{R} \) and \( \tau \subseteq D.\mathcal{U}_t \)

\textbf{Output:} Updated D or fail

1 // Case 0: Failure (waiting forever)
2 if \( p = \infty \) then 
3 return fail 
4 // Case 1: Only contingent timepoints executed
5 if \( \Delta = \text{Wait} \) or \( \Delta = (t, V) \) and \( p < t \) then 
6 \[ HCE(S, D, p, \tau) \]
7 else 
8 // Case 2: Executable timepoint V executes at t 
9 \[ HXE(S, D, t, V) \]
10 // Case 3: CTPs also execute at t
11 if \( t \neq 0 \) then 
12 \[ HCE(S, D, t, \tau) \]
13 D.now := \( p \)
14 return D

\textbf{Update.} The response of the RTE\textsuperscript{*} algorithm to its observation of possible CTP executions is handled by the RTE\textsuperscript{update} algorithm (Algorithm 6). If \( p = \infty \), which can only happen when a Wait decision was made but there were no active contingent links, then the RTE\textsuperscript{*} algorithm would wait forever and, hence, fail (Line 2). Otherwise, \( p < \infty \). If the decision was wait, then one or more contingent TPs must have executed at \( p \) (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 \( p < t \) (Lines 3 to 4).

The HCE algorithm (Algorithm 7) updates D in response to contingent executions as follows. Lines 2 to 3 record that C occurred at p by adding the variable assignment \((C, p)\) to D.f and removing C from D.\mathcal{U}_t. 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 D.Ac\textsuperscript{Wts} 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\textsuperscript{update} (Algorithm 6), the decision is \((t, V)\) and \( p = t \). 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 Ac\textsuperscript{Wts} 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 \( p \) (Line 9).

Table 2 shows sample runs of the RTE\textsuperscript{*} 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 C.
Table 2: Sample runs of the RTE* algorithm on the dispatchable ESTNU from Figure 6(a).

<table>
<thead>
<tr>
<th>Iter.</th>
<th>T(_W)(A)</th>
<th>T(_W)(X)</th>
<th>T(_W)(Y)</th>
<th>Ac(_w)ts(A)</th>
<th>Ac(_w)ts(X)</th>
<th>Ac(_w)ts(Y)</th>
<th>now</th>
<th>Enab(<em>s</em>)</th>
<th>RTED</th>
<th>Obs</th>
<th>Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0, \infty)</td>
<td>[0, \infty)</td>
<td>[0, \infty)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>0</td>
<td>Z := 0</td>
</tr>
<tr>
<td>1</td>
<td>[6, \infty)</td>
<td>[0, \infty)</td>
<td>[0, \infty)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7(_A)</td>
<td>7</td>
<td>A := 7</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>[0, \infty)</td>
<td>[0, \infty)</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16(_C)</td>
<td>0</td>
<td>C := 2</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>[0, 15]</td>
<td>[0, 15]</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>13(_Y)</td>
<td>13</td>
<td>Y := 13</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>[15, 15]</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15(_X)</td>
<td>15</td>
<td>X := 15</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Iter.</th>
<th>T(_W)(A)</th>
<th>T(_W)(X)</th>
<th>T(_W)(Y)</th>
<th>Ac(_w)ts(A)</th>
<th>Ac(_w)ts(X)</th>
<th>Ac(_w)ts(Y)</th>
<th>now</th>
<th>Enab(<em>s</em>)</th>
<th>RTED</th>
<th>Obs</th>
<th>Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0, \infty)</td>
<td>[0, \infty)</td>
<td>[0, \infty)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>0</td>
<td>Z := 0</td>
</tr>
<tr>
<td>1</td>
<td>[6, \infty)</td>
<td>[0, \infty)</td>
<td>[0, \infty)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7(_A)</td>
<td>7</td>
<td>A := 7</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>[0, \infty)</td>
<td>[0, \infty)</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16(_C)</td>
<td>0</td>
<td>C := 2</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>[18, \infty)</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16(_X)</td>
<td>17</td>
<td>C := 17</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>[18, 20]</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>19(_X)</td>
<td>19</td>
<td>X := 19</td>
</tr>
</tbody>
</table>

### RTE* Complexity

The worst-case complexity of the RTE* algorithm is similar to that of the RTE algorithm except for the maintenance of the \(A\) and ?? sets (which is handled by the HCE and HXE algorithms). The \(A\) and ?? sets can also be implemented using min priority queues. Since there are at most \(nk\) wait edges, each of which gets inserted into an \(A\) 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 \(\text{Observe}\_\text{RTE}\) 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 \(S = (\mathcal{T}, \mathcal{C}, \mathcal{L}, \mathcal{C}_\omega)\) be an ESTNU. Every run of the RTE* algorithm on \(S\) corresponds to a run of the RTE algorithm for STNs on some STN projection \(S_\omega\) of \(S\), 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 \(\sigma\), we write \(\sigma(X) = \perp\). In addition, we let \(\max(\sigma) = t_h\) note the time of the latest execution event in \(\sigma\).

**Definition 6** (Pre-history). The pre-history \(\pi_\omega\) 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 \(S_\omega\) respects a pre-history \(\pi_\omega\) if it is consistent with the constraints on the durations specified by \(\pi\).

**Definition 8** (RTE-compliant). A (possibly partial) execution sequence \(\sigma\) is RTE-compliant for an ESTNU \(S\) if it can be generated by some run of the RTE algorithm on every projection \(S_\omega\) that respects the pre-history \(\pi_\omega\).

**Proof.** This proof incrementally analyzes an arbitrary execution sequence generated by the RTE* algorithm on the ESTNU \(S\), 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 \(S\) 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:

\[\text{max}(\sigma) = t_h\]

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

**Definition 9** (Respect). A projection \(S_\omega\) respects a pre-history \(\pi_\omega\) if it is consistent with the constraints on the durations specified by \(\pi\).

**Definition 10** (RTE-compliant). A (possibly partial) execution sequence \(\sigma\) is RTE-compliant for an ESTNU \(S\) if it can be generated by some run of the RTE algorithm on every projection \(S_\omega\) that respects the pre-history \(\pi_\omega\).

**Proof.** This proof incrementally analyzes an arbitrary execution sequence generated by the RTE* algorithm on the ESTNU \(S\), 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 \(S\) 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) $D_f = f$ (i.e., the current, typically partial execution sequences are the same); and

(P2) $f$ is RTE compliant for $S$.

Base Case. $D_f = \emptyset = f$, and $\emptyset$ is trivially RTE-compliant for $S$.

Recursive Case. Suppose (P1) and (P2) hold at the beginning of some iteration. First, note that $D_f = f$ implies that $D \cup D_f = \emptyset$. In the case where these sets are both empty, both algorithms terminate, signaling that $f$ 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 $D \cup D_f = \max(D_f) = \max(f) = \max(f) = \max(D_f)$. Next, we show that $D \cup D_f \cup D_f \cap \emptyset$. This follows because each negative edge in $\emptyset$ is either an ordinary edge or a wait edge, both of which project onto negative edges in every projection. Since $D \cup D_f \cup D_f \cap \emptyset$, only includes executable TPs, the equality holds.

Case 1: $D \cup D_f = \emptyset$. Therefore, $D \cup D_f \cap \emptyset$. Then the RTE algorithm generates a wait decision. Now, $D \cup D_f = \emptyset$ would cause RTE to fail (Algorithm 1, Line 7), contradicting the dispatchability of any STN projection from this point onward. Therefore, $D \cup D_f \cap \emptyset$, 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 $(t, \tau)$, where $\tau \neq \emptyset$.

Now, by (P2), $f$ is RTE-compliant; hence it can be generated by any projection that respects the pre-history $\pi_f$. Next, let $f'$ be the execution sequence obtained by executing the CTPs in $\tau$ at time $t$, and let $\pi_{f'}$ be the corresponding pre-history. Among the projections that respect the pre-history $\pi_{f'}$ are those that also respect $\pi_f$. Since the RTE algorithm, when applied to any of those projections, must execute the CTPs in $\tau$ at time $t$, it follows that $f'$ is RTE compliant for $S$ (i.e., (P2) holds at the start of the next iteration). And since the $\text{HCE}$ algorithm executes the CTPs in $\tau$ at $t$, it follows that (P1) holds at the start of the next iteration. Finally, the other updates done by $\text{HCE}$ 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 $D \cup D_f \cup D_f \cap \emptyset$, is equivalent to RTE’s updating of $D \cup D_f \cup D_f \cap \emptyset$ given that wait edges project onto ordinary negative edges.

Case 2: $D \cup D_f \cap \emptyset \neq \emptyset$. Here, the RTE algorithm (Algorithm 4) would, at Lines 3 to 12, generate an execution decision of the form $(t, \tau)$. Now, for any (executable) $X \in \text{Enabs}_S$, its upper bound is computed based solely on propagations from executed TPs along non-negative edges. Given that $D_f = f$, it follows that $D \cup D_f \cap \emptyset$. Therefore, lines 7 to 13 of the oracle (Algorithm 5) would select an observation of the form $(t, \tau)$, where $\tau \neq \emptyset$.

Base Case. $D \cup D_f \cap \emptyset = \emptyset$, and $\emptyset$ is trivially RTE-compliant for $S$.

Recursive Case. Suppose (P1) and (P2) hold at the beginning of some iteration. First, note that $D \cup D_f \cap \emptyset = \emptyset$ implies that $D \cup D_f \cap \emptyset = \emptyset$. In the case where these sets are both empty, both algorithms terminate, signaling that $f$ 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 $D \cup D_f \cap \emptyset = \max(D_f) = \max(f) = \max(D_f)$. Next, we show that $D \cup D_f \cap \emptyset$. This follows because each negative edge in $\emptyset$ is either an ordinary edge or a wait edge, both of which project onto negative edges in every projection. Since $D \cup D_f \cap \emptyset$, only includes executable TPs, the equality holds.

Case 1: $D \cup D_f \cap \emptyset = \emptyset$. Therefore, $D \cup D_f \cap \emptyset$. Then the RTE algorithm generates a wait decision. Now, $D \cup D_f \cap \emptyset = \emptyset$ would cause RTE to fail (Algorithm 1, Line 7), contradicting the dispatchability of any STN projection from this point onward. Therefore, $D \cup D_f \cap \emptyset$, 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 $(t, \tau)$, where $\tau \neq \emptyset$.

Now, by (P2), $f$ is RTE-compliant; hence it can be generated by any projection that respects the pre-history $\pi_f$. Next, let $f'$ be the execution sequence obtained by executing the CTPs in $\tau$ at time $t$, and let $\pi_{f'}$ be the corresponding pre-history. Among the projections that respect the pre-history $\pi_{f'}$ are those that also respect $\pi_f$. Since the RTE algorithm, when applied to any of those projections, must execute the CTPs in $\tau$ at time $t$, it follows that $f'$ is RTE compliant for $S$ (i.e., (P2) holds at the start of the next iteration). And since the $\text{HCE}$ algorithm executes the CTPs in $\tau$ at $t$, it follows that (P1) holds at the start of the next iteration. Finally, the other updates done by $\text{HCE}$ 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 $D \cup D_f \cap \emptyset$, is equivalent to RTE’s updating of $D \cup D_f \cap \emptyset$ given that wait edges project onto ordinary negative edges.

Case 2: $D \cup D_f \cap \emptyset \neq \emptyset$. Here, the RTE algorithm (Algorithm 4) would, at Lines 3 to 12, generate an execution decision of the form $(t, \tau)$. Now, for any (executable) $X \in \text{Enabs}_S$, its upper bound is computed based solely on propagations from executed TPs along non-negative edges. Given that $D_f = f$, it follows that $D \cup D_f \cap \emptyset$. Therefore, lines 7 to 13 of the oracle (Algorithm 5) would select an observation of the form $(t, \tau)$, where $\tau \neq \emptyset$.
Case 2c: A \((t, \tau)\) observation, where \(\tau \neq 0\). This case is similar to a combination of Case 1 (with \(\rho = t\)) and Case 2b.

**Corollary 1.** An ESTNU \(S\) is dispatchable if and only if every run of the RTE* algorithm on \(S\) outputs a solution for the ordinary constraints in \(S\).

**Proof.** By Theorem 2, \(S\) 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 \(S_\omega\). But by Definitions 4 and 1, \(S\) 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 \(S = (I, C, L, C_w)\) is dispatchable (according to the definition in the literature) if and only if every run of RTE* on \(S\) necessarily satisfies all of the constraints in \(C\) 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


