A Fast Incremental Algorithm for Managing the Execution of Dynamically Controllable Temporal Networks

Luke Hunsberger
Vassar College
Poughkeepsie, NY 12604, USA
hunsberg@cs.vassar.edu

***Abstract*—A Simple Temporal Network with Uncertainty (STNU) is a network of time points and temporal constraints in which the durations of certain temporal intervals—the *contingent links*—are bounded, but not controllable. An STNU is *dynamically controllable* if there is a real-time strategy for executing its non-contingent time points that guarantees the consistency of the network no matter how the durations of the contingent links turn out. Morris presented an $O(N^4)$-time algorithm for determining the dynamic controllability of arbitrary STNUs, where N is the number of time points.**

Morris suggested that additional $O(N^4)$-time computation might be needed to prepare a dynamically controllable network for execution, with all computations done in advance of execution. Instead, this paper shows that an STNU that has passed Morris' algorithm is *already* prepared for execution. The paper presents an incremental, real-time execution algorithm that is guaranteed to successfully execute the time points in a dynamically controllable STNU using $O(N^2)$ space and $O(N^4)$ time. The $O(N^4)$-time computations are not done in advance of execution, but instead are spread out over the entire time that time points in the network are being executed: N iterations of $O(N^3)$ per iteration. Furthermore, the most costly computations—$O(N^3)$ per iteration—are done while waiting for the next execution event to occur, whereas the time-critical computations require only $O(N^2)$ per iteration.

***Keywords*-Temporal Networks, Dynamic Controllability**

I. Background

A *Simple Temporal Network* (STN) is a set of time-point variables together with a set of constraints on those variables. The constraint types include release, deadline and duration constraints. Algorithms for determining the consistency of STNs, incrementally propagating constraints through STNs, and generating execution schedules make STNs useful in domains where computer agents need to plan activities that are subject to temporal constraints [1], [2], [3].

In some domains, the planning agent controls the initiation of actions, but not their durations. A *Simple Temporal Network with Uncertainty* (STNU) models this by including a new kind of constraint, called a contingent link. An important problem for a planning agent is to determine whether an STNU is *dynamically controllable*—that is, whether a real-time, dynamic strategy exists for executing the time points controlled by the agent such that the network is guaranteed to remain consistent no matter how the uncertain durations turn out. Polynomial algorithms for determining the dynamic controllability of STNUs, and for generating the requisite dynamic execution strategies, make STNUs practical for a variety of real-world applications [4], [5], [6].

The rest of this section reviews the relevant definitions and results from prior work on temporal networks and dynamic controllability that are needed for the rest of the paper.

A. Simple Temporal Networks

A Simple Temporal Network is a pair, $(\mathcal{T}, \mathcal{C})$, where $\mathcal{T}$ is a set of time-point variables—or time points (TPs)—and $\mathcal{C}$ is a set of binary constraints, each having the form, $Y - X \leq \delta_{xy}$, for some $X, Y \in \mathcal{T}$ and $\delta_{xy} \in \mathbb{R}$ [1]. A pair of constraints, $Y - X \leq b$ and $X - Y \leq -a$, are often abbreviated as $Y - X \in [a, b]$. Typically, one of the time points in $\mathcal{T}$, called Z, has its value fixed at 0. Binary constraints involving Z are equivalent to unary constraints. For example, $X - Z \leq b$ and $Z - X \leq -a$ are equivalent to $X \in [a, b]$. An STN is called *consistent* if there is a set of values for its time points that satisfy all of its constraints.

Each STN, $(\mathcal{T}, \mathcal{C})$, has a corresponding graph, $(\mathcal{N}, \mathcal{E})$, where the nodes in $\mathcal{N}$ correspond to the time points in $\mathcal{T}$, and the directed edges in $\mathcal{E}$ correspond to the constraints in $\mathcal{C}$. In particular, each constraint, $Y - X \leq b$ in $\mathcal{C}$, corresponds to an edge, $X \xrightarrow{b} Y$ in $\mathcal{E}$. The *all-pairs, shortest-path* matrix for the graph of an STN is called its *distance matrix*, which is often denoted by $\mathcal{D}$.

The time points other than Z are called *executable.* To *execute* a time point, X, at some time t, means to assign the value t to X. This is represented by inserting the constraint, $X \in [t, t]$ (i.e., $X = t$). The executable time points in an STN are presumed to be under the control of some agent operating in real time. At any time t, the agent is presumed to be free to execute any previously unexecuted time point. Once executed, a time point's value is permanently fixed.

B. STNs with Uncertainty

An *STN with Uncertainty* is an STN with a set of special temporal intervals, called *contingent links*, whose durations are beyond the control of the planning agent [4]. 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 contingent link has the

form, (A, x, y, C), where A and C are time points, and $0 < x < y < \infty$. A is called the *activation* time point for the contingent link; C is its *contingent* time point. Certain restrictions on the contingent links (e.g., that two contingent links cannot share the same contingent time point) are captured by defining STNUs recursively, as follows:

- If $(\mathcal{T}, \mathcal{C})$ is an STN, then $(\mathcal{T}, \mathcal{C}, \emptyset)$ is an STNU.
- If $(\mathcal{T}, \mathcal{C}, \mathcal{L})$ is an STNU, $0 < x < y < \infty$, $A \in \mathcal{T}$, and $C \notin \mathcal{T}$, then $(\mathcal{T}', \mathcal{C}', \mathcal{L}')$ is an STNU, where:

$$\begin{aligned} \mathcal{T}' &= \mathcal{T} \cup \{C\} \\ \mathcal{C}' &= \mathcal{C} \cup \{C - A \in [x, y]\} \\ \mathcal{L}' &= \mathcal{L} \cup \{(A, x, y, C)\} \end{aligned}$$

$\mathcal{T}$ can be partitioned into three sets: the contingent time points, the executable time points, and $\{Z\}$. Note that activation time points can be executable or contingent; thus, contingent links can form chains or trees, but not cycles.[1]

In this paper, N denotes the number of time points in an STNU, and K the number of *contingent* time points.

For an STNU, $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L})$, there are two important STNs: $\mathcal{S}_0 = (\mathcal{T}, \mathcal{C})$, which treats contingent links like ordinary constraints; and $\mathcal{S}_x$, called the *AllMax* projection, in which all contingent durations are forced to take on their maximum values. The corresponding graphs are denoted by $\mathcal{G}_0$ and $\mathcal{G}_x$; the distance matrices by $\mathcal{D}_0$ and $\mathcal{D}_x$.

C. Dynamic Controllability

The agent directly controls the execution of the executable time points in an STNU; however, the execution of the contingent time points is beyond the agent's direct control. For example, if (A, x, y, C) is a contingent link for which A is an executable time point, the agent directly controls only the execution of A. Once A has been executed (i.e., once the contingent link has been *activated*), the execution of C is out of the agent's control. Although C is guaranteed to be executed such that $C - A \in [x, y]$, the agent does not get to choose the particular time, but only *observes* the execution of C when it happens. Similar remarks apply to a tree of contingent links with an executable time point at its root.

An STNU is *dynamically controllable* if there exists a dynamic strategy for executing the executable time points that guarantees the consistency of the network, no matter how the durations of the contingent links turn out—within their specified bounds. Crucially, the decisions constituting a *dynamic execution strategy* cannot depend on advance knowledge of the durations of the contingent links. Morris, Muscettola and Vidal—hereinafter MMV—presented a concise semantics for dynamic controllability [4]. This author fixed a technical flaw in their semantics to properly capture the prohibition against decisions based on advance knowledge and to enable a characterization of dynamic execution strategies in terms of *real-time execution decisions* [7].

Real-time execution decisions (RTEDs) have two forms:

- `WAIT`: Wait for some contingent TP to execute.
- (T, χ): If no contingent TPs execute before time T, then execute the (executable) TPs in χ at time T.

The *outcome* of an RTED, which the agent observes in real time, specifies the time points that execute *next.* The outcome of a `WAIT` decision necessarily involves the execution of only contingent time points; the outcome of a (T, χ) decision might involve the execution of contingent time points, executable time points, or both.

D. DC-Checking Algorithms

DC-checking algorithms are algorithms that determine whether STNUs are dynamically controllable. This section summarizes three important DC-checking algorithms.

The MMV DC-checking algorithm: MMV presented a *pseudo-polynomial* DC-checking algorithm based on the generation and propagation of a new kind of constraint, called a *wait* [4]. Each wait has the form: *while the contingent time point C remains unexecuted, the execution of B must wait at least w units after the execution of C's activation time point.* MMV's DC-checking algorithm is sound and complete with respect to the fixed semantics.

The MM DC-checking algorithm: Morris and Muscettola—hereinafter MM—presented the first truly polynomial DC-checking algorithm [5]. It begins by creating a graph, $\mathcal{G}$, that contains all edges from $\mathcal{G}_0$ plus two new kinds of *labeled edges*. In particular, for each contingent link, (A, x, y, C), $\mathcal{G}$ includes the *lower-case* edge, $A \xrightarrow{c:x} C$, and the *upper-case* edge, $A \xleftarrow{C:-y} C$. The lower-case edge represents the possibility that the duration of the contingent link might be its *minimum* value, x; the upper-case edge represents the possibility that the duration might be its *maximum* value, y. Without the labels, these edges would be mutually inconsistent; thus, the labels must be carefully maintained when propagating constraints.

The MM DC-checking algorithm uses the following rules to generate new edges.[2]

(Upper Case) $A \xleftarrow{B:x} C \xleftarrow{y} D$ adds: $A \xleftarrow{B:(x+y)} D$

(Lower Case) $A \xleftarrow{x} C \xleftarrow{c:y} D$ adds: $A \xleftarrow{x+y} D$

(Cross Case) $A \xleftarrow{B:x} C \xleftarrow{c:y} D$ adds: $A \xleftarrow{B:(x+y)} D$

(No Case) $A \xleftarrow{x} C \xleftarrow{y} D$ adds: $A \xleftarrow{x+y} D$

The Lower-Case rule only applies if $x \leq 0$ and $A \neq C$. The Cross-Case rule only applies when $x \leq 0$ and $B \neq C$.

In addition, the following rule, which applies when $z \geq -x$, governs the removal of upper-case labels.

(Label Removal) $B \xleftarrow{b:x} A \xleftarrow{B:z} C$ adds $A \xleftarrow{z} C$

Note that no rule generates new lower-case edges. Thus, an STNU always has exactly K lower-case edges. In contrast, the Upper-Case and Cross-Case rules may generate

[1] A cycle of contingent links would be inherently inconsistent.

[2] The rules are shown using MM's original notation. Caution! The x's and y's here are not necessarily bounds for contingent links. Also, C is only required to be contingent in the Lower-Case and Cross-Case rules, where its activation TP is D and its *lower* bound is y. Finally, in the Upper-Case and Cross-Case rules, B is contingent, with activation time point A.

new upper-case edges. However, the target of an upper-case edge is invariably the activation time point for the corresponding contingent link. Thus, an STNU can have at most KN upper-case edges. MM showed that the upper-case edges generated by their algorithm are equivalent to the *waits* generated by the earlier, MMV algorithm.

The MM DC-checking algorithm uses the rules given above to generate new edges, each of which is inserted into, and propagated through the *AllMax* matrix, $\mathcal{D}_x$. MM showed that, for dynamically controllable networks, no more than N^2 rounds of edge generation are required to generate *all* of the derivable edges. They also showed that if $\mathcal{D}_x$ remains consistent after N^2 such rounds, then the network must be dynamically controllable. The MM algorithm is sound and complete and runs in $O(N^5)$ time.

Morris' DC-checking algorithm: Morris [6] developed a faster, $O(N^4)$-time DC-checking algorithm by focusing on the edges that could be generated from the fixed supply of lower-case edges in an STNU. In particular, he showed how to more efficiently search for relevant applications of the Lower-Case and Cross-Case rules to each lower-case edge. Morris' DC-checking algorithm performs only K rounds of edge generation using these two rules and propagating the corresponding constraints through the *AllMax* matrix, $\mathcal{D}_x$. He proved that if $\mathcal{D}_x$ were still consistent at the end of this process, then the STNU must be dynamically controllable.

The correctness of Morris' algorithm depends on several important properties concerning the generation of new edges from lower-case edges. Since these properties will be needed later on, they are summarized below.[3]

Reduced Distance: The *reduced distance* of a path is the sum of its edge lengths, ignoring any labels. If time point T_i occurs before T_j in a path $\mathcal{P}$, then $\mathcal{D}_{\mathcal{P}}(T_i, T_j)$ denotes the reduced distance from T_i to T_j in $\mathcal{P}$.

Path reductions: A path $\mathcal{P}$ is called *reducible* if it contains a pair of edges to which one of the edge-generation rules can be applied.[4] Replacing that pair of edges with the newly generated edge results in a new path $\mathcal{P}'$. The process of moving from $\mathcal{P}$ to $\mathcal{P}'$ is called a *reduction.* By inspection, each edge-generation rule preserves the reduced distance; thus, so does any path reduction.

Semi-reducible paths: A path is called *semi-reducible* if it can be transformed by a sequence of reductions into a path without any *lower-case* edges. Morris proved that an STNU is dynamically controllable if and only if it does not have a semi-reducible negative cycle (i.e., a semi-reducible path that forms a loop whose reduced distance is negative). Morris' DC-checking algorithm searches for semi-reducible negative cycles; however, this search is made more efficient by focusing on *extension sub-paths.*

[3]For mathematical convenience, Morris assumes that: (1) agents can respond instantaneously to contingent executions; and (2) contingent links can have lower bounds of zero. He transforms STNUs into a *normal form,* where each contingent link has a lower bound of 0. This paper does not make these assumptions or transformations. Thus, the summary of Morris' work given here contains some slight differences (e.g., $\leq$ instead of $<$ in places). As Morris noted, his assumptions are not necessary. This author prefers to avoid the assumption of instantaneous reactivity.

[4]A simpler characterization applies to the Label-Removal rule.

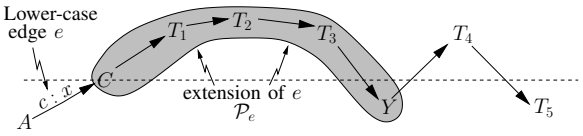

Figure 1. An extension sub-path

Extension sub-paths: Suppose e is a lower-case edge, $A \xrightarrow{c:x} C$, within a path $\mathcal{P}$. The *extension* of e in $\mathcal{P}$, if it exists, is a sub-path $\mathcal{P}_e$ of $\mathcal{P}$ such that:

- $\mathcal{P}_e$ immediately follows the edge e in $\mathcal{P}$ and, thus, is a sub-path from C to some other time point Y;
- $\mathcal{D}_{\mathcal{P}}(C, T_i) > 0$, for each *interior* time point T_i in $\mathcal{P}_e$;
- $\mathcal{D}_{\mathcal{P}}(C, Y) \leq 0$.

The extension sub-path is illustrated in Fig. 1, where the vertical position of each time point T_i represents the reduced distance from C to T_i—which remains positive until the last edge in the extension sub-path.

The extension sub-path, $\mathcal{P}_e$, is important because if it can be reduced to a single edge, then that edge could be used to "reduce away" the lower-case edge e, using either the Lower-Case or Cross-Case rule. That could only be prevented if $\mathcal{P}_e$ reduced to an upper-case edge from the *same* contingent link as e—because then the Cross-Case rule could not be used.[5] That could only happen if $\mathcal{P}_e$ contained an upper-case edge from the same contingent link as e, a situation that Morris calls a *breach.* Morris proved that if an STNU has a semi-reducible negative cycle, then it has a *breach-free* semi-reducible negative cycle.

In addition, Morris showed that if a path $\mathcal{P}$ contains extension sub-paths from two lower-case edges, then those extension sub-paths must either be disjoint or nested (i.e., one fully inside the other). He then used that to prove that if an STNU has a semi-reducible negative cycle, it must have a breach-free semi-reducible negative cycle in which the depth of nesting of extension sub-paths is at most K.

Thus, Morris' DC-checking algorithm performs K rounds of searching through breach-free extensions of lower-case edges, looking for applications of the Lower-Case or Cross-Case rules. Each round effectively increments the depth of nesting of extension sub-paths. If, after receiving all of the edges generated by K such rounds, the *AllMax* matrix, $\mathcal{D}_x$, is still consistent, then the network is necessarily dynamically controllable.

E. DC-Checking vs. Execution

A DC-checking algorithm is only responsible for determining whether an STNU is dynamically controllable.

[5]Recall the restriction $B \neq C$ in the Cross-Case rule.

That is, it need only ensure the *existence* of a dynamic execution strategy; it need not construct one. However, in successful instances (i.e., when the network in question turns out to be dynamically controllable), both the MMV and MM DC-checking algorithms generate *all* of the edges that are derivable from the edge-generation rules.[6] Thus, both of these DC-checking algorithms also effectively prepare the network for execution. In particular, MMV showed that this full complement of edges can be used as the basis for a real-time execution algorithm—henceforth called the MMV-EX algorithm—that guarantees the consistency of the STNU no matter how the contingent durations turn out. Unfortunately, the information used by this algorithm is generated in $O(N^5)$ time by the MM DC-checking algorithm.

In contrast, Morris' faster, $O(N^4)$-time DC-checking algorithm typically does *not* generate all of the edges that are derivable from the edge-generation rules. Instead, it focuses on "reducing away" the lower-case edges. Thus, for any STNU that passes Morris' algorithm, it seemed plausible that extra work might be required to prepare the network for execution. Toward that end, Morris briefly sketched an extra $O(N^4)$-time procedure to generate the rest of the derivable upper-case and ordinary edges—in advance of execution.[7] The next section demonstrates that this is not necessary.

II. The NEW-EX Execution Algorithm

This section presents a new execution algorithm, called *NEW-EX,* that takes as its starting point an STNU that has passed Morris' DC-checking algorithm. The algorithm's overall computational complexity is $O(N^4)$. However, this $O(N^4)$-time computation is not done in advance of execution; instead, it is spread out over the entire time that time points in the network are being executed: N iterations at $O(N^3)$ per iteration. Furthermore, the most expensive, $O(N^3)$ computations can be done while the agent is waiting for the next execution event to occur, whereas the time-critical computations require only $O(N^2)$ per iteration.

The NEW-EX algorithm demonstrates that it is not necessary to generate all of the additional upper-case and ordinary edges in advance of execution, as suggested by prior work. Instead, a network that passes Morris' DC-checking algorithm is, in effect, *already* prepared for execution.

The NEW-EX algorithm begins with the network existing at the end of Morris' DC-checking algorithm. The ordinary and upper-case edges in that network together comprise the *core edges.* The NEW-EX algorithm maintains two distance matrices: $\mathcal{D}_0$, which is initialized with only the *ordinary* core edges; and $\mathcal{D}_x$, which is initialized with *all* of the core edges. During the execution phase, whenever any (contingent or non-contingent) time point executes, both of these matrices are updated accordingly. In addition, whenever any *contingent* time point executes, the NEW-EX algorithm, in effect, *removes* the upper-case edges associated with that time point from the network. This updating of $\mathcal{D}_x$ is performed by first resetting $\mathcal{D}_x$ to a copy of $\mathcal{D}_0$, which contains no upper-case edges, and then re-inserting the upper-case core edges for the *unexecuted* contingent time points.

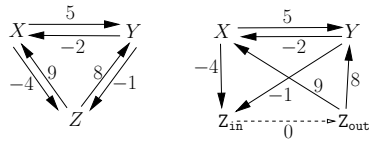

Figure 2. Splitting Z into $\mathtt{Z_{in}}$ and $\mathtt{Z_{out}}$

The core set of edges: Let $\mathcal{S}$ be an STNU and $\mathcal{G}$ its corresponding graph, which includes the *original* lower-case and upper-case edges for each contingent link. Let the *core set of edges* denote the ordinary and upper-case edges from $\mathcal{G}$ together with those generated by Morris' algorithm. Note that the core set of edges does not include any lower-case edges. The consistency check in Morris' algorithm consists of inserting the core edges—minus their labels—into the *AllMax* matrix, $\mathcal{D}_x$, and then propagating them. Note that $\mathcal{D}_x$ does not distinguish upper-case and ordinary edges.

The main insight behind the NEW-EX algorithm is that prior to the execution of a contingent time point, C, it is not necessary to distinguish upper-case edges associated with C and ordinary edges. As shown in prior work [4], each upper-case edge, $B \xrightarrow{C:-w} A$, is equivalent to a constraint of the form, "as long as C remains unexecuted, B must not be executed until at least w units after the execution of A." Thus, prior to the execution of C, this upper-case edge is indistinguishable from the ordinary edge, $B \xrightarrow{-w} A$. Thus, prior to the execution of C, $\mathcal{D}_x$ contains the information needed by the execution algorithm concerning C.

Once C executes, the upper-case edges associated with C are no longer needed. Thus, if $\mathcal{D}_x$ can be updated during the course of execution so that its entries reflect only that which can be derived from the core upper-case edges associated with *unexecuted* contingent time points, then $\mathcal{D}_x$ will always have exactly what is needed by the execution algorithm. The required updating of $\mathcal{D}_x$ involves effectively removing the associated core upper-case edges from $\mathcal{D}_x$ whenever a contingent time point executes.

A. Initializing the NEW-EX Algorithm

The NEW-EX algorithm makes one basic, but very important modification to the temporal network that dramatically reduces the amount of subsequent constraint propagation during execution: it splits the Z time point into two time points, $\mathtt{Z_{in}}$ and $\mathtt{Z_{out}}$, as described in earlier work [2], and as illustrated in Fig. 2. As a result of this change, all edges that formerly had Z as their destination now have $\mathtt{Z_{in}}$ as

[6] Recall that MM showed that the upper-case edges generated by their algorithm are equivalent to the waits generated by the MMV algorithm.

[7] Morris did not include implementation details or a proof of correctness.

IF $\mathcal{U}_x$ and $\mathcal{U}_c$ are both empty, THEN done, ELSE:

1. Generate the next real-time execution decision, RD.
2. Launch the real-time execution decision, RD.
3. If necessary, prepare the helper matrix, $\mathcal{D}'_x$.
4. Observe execution outcome: $(\texttt{NOW}', \texttt{NewExec})$.
5. If any contingent time points executed, $\mathcal{D}_x := \mathcal{D}'_x$.
6. Add (and propagate) execution constraints for newly executed time points and constraints that unexecuted time points occur at or after $\texttt{NOW}'$.
7. If any contingent time points executed, update $\mathcal{D}_x$.
8. Go to next iteration with $\texttt{NOW} := \texttt{NOW}'$.

Figure 3. One iteration of NEW-EX algorithm

their destination; and all edges that formerly had Z as their source now have $\texttt{Z}_{\texttt{out}}$ as their source. This change would have little effect if an edge from $\texttt{Z}_{\texttt{in}}$ to $\texttt{Z}_{\texttt{out}}$ of length 0 were also added to the network, depicted by a dashed line in the figure. However, this edge is purposely left out to eliminate the possibility of propagating constraints through (what used to be) Z. Although the omission of this edge typically changes the shortest-path information stored in the corresponding distance matrix, the original shortest-path information is easily retrieved from the new distance matrix. In particular, if $\mathcal{D}^*_0$ is the distance matrix for the modified network, then the original entry, $\mathcal{D}_0(X, Y)$, is given by:

$$\mathcal{D}_0(X,Y) \;=\; \min\{\mathcal{D}^*_0(X,Y),\; \mathcal{D}^*_0(X,\texttt{Z}_{\texttt{in}}) + \mathcal{D}^*_0(\texttt{Z}_{\texttt{out}},Y)\}$$

The elimination of propagation through Z is particularly helpful during execution, since execution constraints invariably involve Z. The rest of this paper presumes that this modification has been made to all networks.

The NEW-EX algorithm is initialized as follows.

- The following hash tables are initialized:
  $\mathcal{U}_x$: the unexecuted executable time points
  $\mathcal{U}_c$: the unexecuted contingent time points
  $\mathcal{A}^+$: the activated, but unexecuted contingent TPs
  $\mathcal{A}^-$: the unactivated contingent time points
  $\mathcal{EX}$: the executed time points.
- The strongest ordinary edges generated by Morris' DC-checking algorithm—at most N^2—are added to the distance matrix, $\mathcal{D}_0$, and fully propagated using the $O(N^3)$ Floyd-Warshall algorithm [8].
- The *AllMax* distance matrix, $\mathcal{D}_x$, is that which exists at the end of Morris' DC-checking algorithm.
- The upper-case edges generated by Morris' algorithm are collected in a K-by-N matrix, UC. In particular, each upper-case edge, $B \xrightarrow{C:-w} A$, yields the entry $\texttt{UC}(C, B) = -w$. The UC matrix does not change during execution; it is used only to update $\mathcal{D}_x$.

B. The Iterative Execution Algorithm

Fig. 3 summarizes one iteration of the NEW-EX algorithm. The individual steps are discussed in further detail below. The first iteration begins with NOW set to 0.

$m := \min\{a + x \mid C \in \mathcal{A}^+\}$
  (A is the activation TP for C, executed at a)
IF $m \leq T$,
  FOR $i = 1$ to N
    FOR $j = 1$ to N
      $\mathcal{D}'_x(i,j) := \mathcal{D}_0(i,j)$
  FOR EACH $C \in \mathcal{A}^-$
    FOR $j = 1$ to N
      IF $\texttt{UC}(C,j) < \mathcal{D}'_x(j,A)$
          (A is the activation TP for C)
        $\mathcal{D}'_x(j,A) := \texttt{UC}(C,j)$
  Run Floyd-Warshall on $\mathcal{D}'_x$

Figure 4. Pseudo-code for Step 3

Step 1 involves generating the next real-time execution decision (RD). In pseudo-code, it looks like this:

IF $\mathcal{U}_x$ empty, THEN $RD := \texttt{WAIT}$
ELSE
$$T := \min\{-\mathcal{D}_x(X, \texttt{Z}_{\texttt{in}}) \mid X \in \mathcal{U}_x\}$$
$$\chi := \{X \in \mathcal{U}_x \mid -\mathcal{D}_x(X, \texttt{Z}_{\texttt{in}}) = T\}$$
$$RD := (T, \chi)$$

If only contingent time points remain to be executed, then the decision is WAIT. Otherwise, the *AllMax* matrix, $\mathcal{D}_x$, is used to determine the lower bounds for each of the as-yet-unexecuted executable time points. The minimum such value is called T. χ is then the set of the as-yet-unexecuted executable time points whose lower bound is T. The generated decision can be glossed as: "If nothing happens before time T, then execute the time points in χ at time T."

Step 2 of the algorithm is to *launch* the decision—which simply means that the agent commits to it. In the case of a WAIT decision, the agent waits to see which contingent time point(s) will execute next. For a (T, χ) decision, the agent waits to see whether any contingent time points will happen to execute before time T.

While waiting to see what happens next, the agent carries out Step 3, which involves creating a helper matrix, $\mathcal{D}'_x$. This matrix will be needed later on should one or more contingent time points happen to execute. The pseudo-code for Step 3 is given in Fig. 4. It begins by computing m, the earliest time at which a currently *activated* contingent time point might execute. If $m \leq T$, then it is possible that the next execution event will involve contingent time points, in which case the helper matrix would be needed. The helper matrix starts out as a fresh copy of $\mathcal{D}_0$. Then, for each currently *unactivated* contingent time point, the corresponding upper-case edges—from the UC matrix—are added to $\mathcal{D}'_x$. The reason is that a currently unactivated contingent time point *cannot* execute next; thus, it will still be unexecuted in the next iteration; thus, its upper-case edges need to be included in $\mathcal{D}'_x$. After all such edges have been

FOR EACH $X \in$ NewExec and $\mathcal{D} \in \{\mathcal{D}_0, \mathcal{D}_x\}$:
  LinearUpdate$(\mathcal{D}, \mathtt{Z_{out}}, X, \mathtt{NOW}')$
  LinearUpdate$(\mathcal{D}, X, \mathtt{Z_{in}}, -\mathtt{NOW}')$
  Remove X from $\mathcal{U}_x$ or $\mathcal{U}_c$
  Add X to $\mathcal{EX}$ with value NOW′
  IF X is an activation time point for some contingent time point C, then move C from $\mathcal{A}^-$ to $\mathcal{A}^+$
  FOR EACH $Y \in \mathcal{U}_x \cup \mathcal{U}_c$ and $\mathcal{D} \in \{\mathcal{D}_0, \mathcal{D}_x\}$:
    LinearUpdate$(\mathcal{D}, Y, \mathtt{Z_{in}}, -\mathtt{NOW}')$

Figure 5. Pseudo-code for Step 6

LinearUpdate$(\mathcal{D}, \mathtt{Z_{out}}, J, w)$
  $\mathcal{D}(\mathtt{Z_{out}}, J) := w$
  FOR $k = 1$ to N
    IF $w + \mathcal{D}(J,k) < \mathcal{D}(\mathtt{Z_{out}}, k)$
      $\mathcal{D}(\mathtt{Z_{out}}, k) := w + \mathcal{D}(J,k)$

Figure 6. Pseudo-code for LinearUpdate

inserted, the Floyd-Warshall algorithm is used to propagate them throughout the matrix. Further updating of this matrix is postponed until Step 7, at which time it will be known whether any contingent time points did in fact execute. The $O(N^3)$ Floyd-Warshall algorithm dominates Step 3.

In Step 4, the agent simply observes the next execution outcome, which involves the simultaneous execution of one or more time points. NOW′ is the time of the execution event; NewExec is the set of newly executed time points. If the decision was WAIT, then NewExec will contain only contingent time points, executed at NOW′. If the decision was (T, χ), then there are three possibilities [7]:

- NOW′ $< T$ and NewExec $\cap\, \chi = \emptyset$
- NOW′ $= T$ and NewExec $= \chi$
- NOW′ $= T$ and $\chi \subset$ NewExec

In the first case, one or more contingent time points executed before time T; so, the time points in χ were not executed. In the second and third cases, nothing happened *before* time T; so, the time points in χ were executed at time T. The third case includes the (unlikely) possibility that one or more contingent time points also happened to execute precisely at time T.

Step 5 involves checking whether any contingent time points actually executed. If so, then $\mathcal{D}_x$ is replaced by the helper matrix, $\mathcal{D}'_x$, computed in Step 3.

Step 6 carries out the propagation of constraints in response to the new execution event. In particular, each newly executed time point X is constrained by $X =$ NOW′, and each still-unexecuted time point Y is constrained by $Y \geq$ NOW′. Since these constraints involve $\mathtt{Z_{in}}$ and $\mathtt{Z_{out}}$, a linear-time updating function, LinearUpdate, can be used. (This is one of the major advantages derived from splitting Z into $\mathtt{Z_{in}}$ and $\mathtt{Z_{out}}$.)

The pseudo-code for the LinearUpdate function for a

IF NewExec contains contingent time points:
  Vec := new vector of length N
  FOR $j = 1$ to N
    Vec$[j] := \mathcal{D}_x(j, \mathtt{Z_{in}})$
  FOR EACH $C \in \mathcal{A}^+$ with activation time point A
    $a :=$ execution time of A
    FOR $j = 1$ to N
      IF UC$(i,j) - a <$ Vec$[j]$
        THEN Vec$[j] :=$ UC$(i,j) - a$
  FOR $j = 1$ to N
    LinearUpdate$(\mathcal{D}_x, j, \mathtt{Z_{in}}, \mathtt{Vec}[j])$

Figure 7. Pseudo-code for Step 7

constraint of the form, $J - \mathtt{Z_{out}} \leq w$, is shown in Fig. 6. The code for a constraint, $\mathtt{Z_{in}} - I \leq -w$, is analogous. Since Step 6 requires at most $4N$ calls to LinearUpdate, the computations for Step 6 are bounded by $O(N^2)$.

Step 7 completes the updating of $\mathcal{D}_x$ for the next iteration, if necessary. Fig. 7 contains the relevant pseudo-code. Notice that $\mathcal{D}_x$ only needs updating if one or more contingent time points actually executed. In that case, the upper-case edges (from the UC matrix) corresponding to any *activated,* but unexecuted contingent time points, must be added to $\mathcal{D}_x$. (The upper-case edges for unactivated contingent time points were already added during Step 3.) Because the activation time points for these contingent links have already executed, they are, in effect, *rigid* with the time point Z (as represented by the two time points, $\mathtt{Z_{in}}$ and $\mathtt{Z_{out}}$) [9]. Thus, instead of adding edges aimed at the activation time point, the algorithm adds equivalent edges aimed at $\mathtt{Z_{in}}$. Thus, these updates can be performed by the LinearUpdate function. Even better, although there could be, in principle, up to KN such edges, the target is always $\mathtt{Z_{in}}$; thus, it suffices to keep track of only the strongest such updates for each of the N time points in the network. This is okay since $\mathcal{D}_x$ does not distinguish upper-case edges from different contingent time points. A vector of length N is used to accumulate the strongest such constraints. Afterward, at most N of the linear updates are performed, for a total cost of $O(N^2)$ for Step 7.

C. Complexity Analysis

Since at least one time point gets executed during each iteration, the NEW-EX algorithm completes at most N iterations. The dominating computation is Step 3, the creation of the helper matrix, which takes $O(N^3)$ time due to the use of Floyd-Warshall. No other step has complexity more than $O(N^2)$. Thus, the overall complexity of the algorithm is $O(N^4)$. Since it maintains two distance matrices, the matrix of upper-case edges, and several linear-sized hash tables, the algorithm's space complexity is $O(N^2)$.

D. Correctness of the NEW-EX Algorithm

For an STNU that passes Morris' DC-checking algorithm, the NEW-EX algorithm guarantees the consistency of the network throughout execution, no matter how the contingent durations turn out—within their specified bounds. The guarantee stems from the facts proven below. For convenience, in all that follows, the reduced length of a path $\mathcal{P}$ is denoted by $|\mathcal{P}|_r$; $\mathcal{G}$ denotes the graph containing the original ordinary edges from the STNU, together with the lower-case and upper-case edges inserted *before* the application of any edge-generation rules; and $\mathcal{G}_c$ contains, in addition, the ordinary and upper-case edges generated by Morris' algorithm (i.e., $\mathcal{G}_c$ contains the *core* set of edges).

Fact 0: If $\mathcal{G}$ has a semi-reducible path $\mathcal{P}$ from A to B, then it has a breach-free semi-reducible path $\mathcal{P}'$ from A to B in which the depth of nesting of extension sub-paths is at most K, and such that $|\mathcal{P}'|_r \leq |\mathcal{P}|_r$.

Proof 0: Morris proved that if $\mathcal{G}$ has a semi-reducible negative cycle, $\mathcal{P}$, then it has a semi-reducible negative cycle, $\mathcal{P}'$, that is breach-free and in which the depth of nesting of extension sub-paths is at most K. However, his proof did not depend on the path in question being a cycle. Thus, it applies equally well to arbitrary semi-reducible paths. Morris showed that $|\mathcal{P}'|_r \leq |\mathcal{P}|_r$, but only used that fact to ensure that if $\mathcal{P}$ were a negative cycle, then $\mathcal{P}'$ would also be a negative cycle. ■

Fact 1: Let E be any edge that is *derivable* from edges in $\mathcal{G}$ using the edge-generation rules. Then $\mathcal{G}_c$ has a path, $\mathcal{P}$, with the same endpoints as E, such that $|\mathcal{P}|_r \leq |E|_r$. We shall call $\mathcal{P}$ the *associated core path* for E.

Proof 1: Let E be as given above. Let X and Y be the starting and ending time points for E. Since the edge-generation rules only generate upper-case or ordinary edges, E must be an upper-case or ordinary edge and, thus, constitutes a semi-reducible path. Thus, by Fact 0, $\mathcal{G}$ has a semi-reducible path $\mathcal{P}'$ from X to Y in which the depth of nesting of extension sub-paths is at most K, and such that $|\mathcal{P}'|_r \leq |E|_r$. Since Morris DC-checking algorithm searches all such extension sub-paths, any lower-case edges in $\mathcal{P}'$ can be "reduced away" to ordinary or upper-case edges generated by Morris' algorithm. Let $\mathcal{P}$ be the path that results from "reducing away" all of the lower-case edges in this way. Since all reductions preserve reduced distance, $|\mathcal{P}|_r = |\mathcal{P}'|_r \leq |E|_r$. Furthermore, all edges in $\mathcal{P}$ are either original edges from $\mathcal{G}$ or edges generated by Morris' algorithm (i.e., $\mathcal{P}$ contains only core edges). ■

Fact 2: The constraints enforced by the NEW-EX execution algorithm are at least as strong as those enforced by the MMV-EX execution algorithm.

Proof 2: The MMV-EX algorithm enforces constraints corresponding to: (1) ordinary edges from $\mathcal{G}$; (2) ordinary edges added during execution; (3) ordinary edges derived from the edge-generation rules; and (4) upper-case edges derived from the edge-generation rules—but only for *unexecuted* contingent time points. The NEW-EX algorithm also enforces constraints for all edges in categories 1 and 2, but only for the *core* edges from categories 3 and 4.

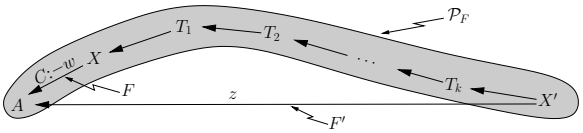

Figure 8. Reducing $\mathcal{P}_F$ to the edge F'

Let E be any derivable ordinary edge (i.e., any edge from category 3). Let $\mathcal{P}$ be the associated core path for E. By construction, there must be a sequence of reductions, $\mathcal{R}$, that transforms $\mathcal{P}$ to E. If there are any upper-case edges in $\mathcal{P}$, they must all eventually be "reduced away" by the reductions in $\mathcal{R}$. Let F be the first upper-case edge, $A \xleftarrow{C:-w} X$, from $\mathcal{P}$ that gets reduced in this way; let F' be the resulting ordinary edge, $A \xleftarrow{z} X'$. Let $\mathcal{P}_F$ be the corresponding sub-path of $\mathcal{P}$ that reduces to the edge F', as illustrated in Fig. 8. Since F is the first upper-case edge to be reduced away, all of the other edges in $\mathcal{P}_F$ must be ordinary edges. In addition, the last reduction that generates F' must be an instance of the Label-Removal rule:

$$C \xleftarrow{c:x} A \xleftarrow{C:z} X' \text{ adds } A \xleftarrow{z} X'$$

(The lower-case edge does not belong to the path $\mathcal{P}_F$.) Thus, the length of F' (i.e., z) must satisfy $z \geq -x$, where x is the minimum value for the contingent link from A to C.

Now consider the constraint, $A - X' \leq z$, represented by the edge F'. Since all of the edges in $\mathcal{P}_F$ are core edges, all of the ordinary edges in $\mathcal{P}_F$ are enforced by the NEW-EX algorithm. The only other edge in $\mathcal{P}_F$ is F; however, this edge is only enforced by the NEW-EX algorithm while C remains unexecuted. Since A necessarily executes before C, there are only two relevant cases to consider: (1) X' executes before C; and (2) X' executes after C (or simultaneously with C). In the first case, the execution times of A and X' are fixed before C executes (i.e., while the edge F is still being enforced by the NEW-EX algorithm). Since F only involves A and X', if it is satisfied when A and X' are executed, it is forever satisfied. In the second case, X' executes after C. But then, $X' \geq C \geq A + x \geq A - z$, since $C - A \geq x$ and $z \geq -x$. Thus, $A - X' \leq z$, as required by F'. Thus, in either case, the constraint corresponding to the edge F' is enforced by the NEW-EX algorithm.

Now, if $\mathcal{P}$ contains any other upper-case edges, they too can be "reduced away" in a similar fashion, generating an ordinary edge that is enforced by the NEW-EX algorithm. When all of the upper-case edges have been reduced away, the result is a path $\mathcal{P}'$ that contains: (1) ordinary edges generated by the "reducing away" process just described; and (2) *core* ordinary edges. Thus, all edges in $\mathcal{P}'$ are enforced by the NEW-EX algorithm and $|\mathcal{P}'| = |\mathcal{P}|_r \leq |E|_r$. Thus,

the NEW-EX algorithm enforces a constraint that is at least as strong as that represented by E.

Next, suppose that E is a category 4 edge (i.e., any *derivable* upper-case edge) whose corresponding contingent time point is C. Then the MMV-EX algorithm only enforces E while C is unexecuted. As before, let $\mathcal{P}$ be the core path for E. If $\mathcal{P}$ contains any other core upper-case edges for C, then those edges are enforced by the NEW-EX algorithm whenever E is enforced by the MMV-EX algorithm. If $\mathcal{P}$ contains any upper-case edges corresponding to some *other* contingent time point, C', then the same line of argument used for the case where E was an ordinary edge can be used to reduce away those upper-case edges from $\mathcal{P}$ such that the resulting ordinary edges will also be enforced by the NEW-EX algorithm. Thus, while C is unexecuted, the NEW-EX algorithm enforces a constraint that is at least as strong as that represented by E. ■

Corollary A: Since the edges enforced by the NEW-EX algorithm are a subset of those enforced by the MMV-EX algorithm, the constraints enforced by the MMV-EX algorithm are at least as strong as those enforced by the NEW-EX algorithm. Thus, in view of Fact 2, the sets of constraints enforced by the two algorithms are equivalent.

Corollary B: Since the MMV-EX and NEW-EX algorithms both execute executable time points as soon as they have satisfied all of their lower-bound constraints, whether derived from ordinary edges or upper-case edges associated with unexecuted contingent time points, the execution decisions generated by the two algorithms must be the same. Since the MMV-EX algorithm guarantees the consistency of the network throughout the execution, then so too does the NEW-EX algorithm.

III. Conclusions and Future Work

This paper presented an $O(N^4)$-time incremental execution algorithm that guarantees the successful execution of any STNU that has passed Morris' $O(N^4)$-time DC-checking algorithm. The NEW-EX algorithm makes the same execution decisions as those generated by the MMV-EX algorithm, but without requiring in advance the complete set of derivable edges generated by the $O(N^5)$-time MM DC-checking algorithm. Furthermore, the NEW-EX algorithm's $O(N^4)$-time computations are spread out over the entire time the network is being executed: N iterations at $O(N^3)$ per iteration. The most expensive $O(N^3)$ computations can be done while the agent waits for the next execution event to occur; the time-critical computations are only $O(N^2)$ per iteration.

The careful distinction between upper-case and ordinary edges that is maintained by the MMV and MM algorithms is useful for managing the subsequent execution, but comes at too high a computational cost. The NEW-EX algorithm avoids this distinction and manages the execution incrementally, for a total cost of $O(N^4)$.

Another corollary to Fact 2 is that the *AllMax* matrices computed by the three DC-checking algorithms are identical. Each contains shortest-path information for semi-reducible paths in $\mathcal{G}$. The consistency of these matrices is *the* critical factor because, as Morris proved, an STNU is dynamically controllable if and only if it has no negative semi-reducible cycles. The incrementally updated matrix used by the NEW-EX algorithm also contains shortest-path information for semi-reducible paths, but only for those whose upper-case edges correspond to unexecuted contingent time points.

In prior work, Morris sketched an $O(N^4)$ procedure for generating, in advance of execution, all of the upper-case and ordinary edges that are derivable from the MM edge-generation rules. As shown by MMV, these edges are sufficient to generate a dynamic execution strategy for a dynamically controllable network. It is left to future work to flesh out the details of Morris' proposed algorithm, prove its correctness, and then compare the real-time performance of that algorithm with the incremental NEW-EX algorithm presented in this paper.

References

[1] R. Dechter, I. Meiri, and J. Pearl, "Temporal constraint networks," *Artificial Intelligence*, vol. 49, pp. 61–95, 1991.

[2] L. Hunsberger, "A practical temporal constraint management system for real-time applications," in *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI-2008)*, M. Ghallab, C. Spyropoulos, N. Fakotakis, and N. Avouris, Eds. Amsterdam: IOS Press, 2008.

[3] N. Muscettola, P. Morris, B. Pell, and B. Smith, "Issues in temporal reasoning for autonomous control systems," in *Proceedings of the Second International Conference on Autonomous Agents*. ACM, 1998, pp. 362–368.

[4] P. Morris, N. Muscettola, and T. Vidal, "Dynamic control of plans with temporal uncertainty," in *Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, 2001, pp. 494–499.

[5] P. H. Morris and N. Muscettola, "Temporal dynamic controllability revisited," in *Twentieth National Conference on Artificial Intelligence (AAAI-2005)*, 2005, pp. 1193–1198.

[6] P. Morris, "A structural characterization of temporal dynamic controllability," in *Principles and Practice of Constraint Programming (CP 2006)*, ser. Lecture Notes in Computer Science. Springer, 2006, vol. 4204, pp. 375–389.

[7] L. Hunsberger, "Fixing the semantics for dynamic controllability and providing a more practical characterization of dynamic execution strategies," in *Proceedings of the 16th International Symposium on Temporal Representation and Reasoning (TIME-2009)*. IEEE Computer Society, 2009.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2009.

[9] A. Gerevini, A. Perini, and F. Ricci, "Incremental algorithms for managing temporal constraints," IRST, Tech. Rep. IRST-9605-07, 1996.