Reformulating Temporal Plans For Efficient Execution

Nicola Muscettola
Recom Technologies.
NASA Ames Research Center
Moffett Field, CA 94035
mus@ptolemy.arc.nasa.gov

Paul Morris
Caelum Research.
NASA Ames Research Center
Moffett Field, CA 94035
pmorris@ptolemy.arc.nasa.gov

Ioannis Tsamardinos
Intelligent Systems Program
University of Pittsburgh
Pittsburgh, PA 15260
tsamard@cs.pitt.edu

Abstract

The Simple Temporal Network formalism permits significant flexibility in specifying the occurrence time of events in temporal plans. However, to retain this flexibility during execution, there is a need to propagate the actual execution times of past events so that the occurrence windows of future events are adjusted appropriately. Unfortunately, this may run afoul of tight real-time control requirements that dictate extreme efficiency. The performance may be improved by restricting the propagation. However, a fast, locally propagating, execution controller may incorrectly execute a consistent plan. To resolve this dilemma, we identify a class of *dispatchable* networks that are guaranteed to execute correctly under local propagation. We show that every consistent temporal plan can be reformulated as an equivalent dispatchable network, and we present an algorithm that constructs such a network. Moreover, the constructed network is shown to have a minimum number of edges among all such networks. This algorithm will be flown on an autonomous spacecraft as part of the Deep Space 1 Remote Agent experiment.

1 Introduction

When designing and implementing control systems operating in a physical world it is important to correctly deal with the metric nature of time. For example, deadlines are typically upper bounds on the value of the occurrence time of certain events (e.g., end of a task). The control system can guarantee a correct execution only if specified time constraints are satisfied for any possible execution. We are interested in the class of high-level control architectures that distinguish between a *deliberative* layer, or *planner*, and a *reactive* layer, or *executive* [10, 1, 13, 6, 11, 9, 2].

This paper is concerned with the properties that must be satisfied by temporally flexible plans in order to be correctly executed by a simplified, fast execution algorithm. The speed of an execution algorithm is central to ensuring that a plan can be robustly executed in real-time, a condition of crucial importance in mission critical applications such as autonomous spacecraft operations [10] and avionics control systems [3]. Unlike fixed time schedules, temporally flexible plans allow an executive to seamlessly adjust to delays and fluctuations of task durations. However, the cost of this flexibility is that the executive must constantly adjust the plan during execution by performing some amount of constraint propagation. The time spent doing this propagation adds to the total time needed to start or end any task in the plan. The latter time is equivalent to the intrinsic time uncertainty on the exact time of occurrence of any event in the plan [7, 8]. The more precise we want the execution of a plan to be, the less propagation an execution algorithm should perform. In this paper, we precisely define fast execution by giving a simple controller algorithm and we describe *dispatchability*, a formal property that identifies whether a plan is amenable to fast execution or not. We also discuss how a non-dispatchable plan can be transformed in polynomial time into a dispatchable plan, and we show that the resulting plan has the desirable property of being minimal in the number of edges among all dispatchable plans.

2 Dispatching Executions

The type of plan that we are interested in is a *temporal plan*, i.e., a partial order of tasks with metric time information. We refer to the start and end times of a task as two separate *events* or *timepoints*. A

plan satisfies the following conditions: (1) for each task, the start and end events must be separated by a non-negative duration $[d, D]$; (2) additional separation constraints $[s, S]$ may be specified between the start and/or the end of any two tasks. A temporal bound-constraint $[b, B]$ (either duration or separation) from an event A to an event B constrains the possible values of the times of occurrence of A and B, T_A and T_B respectively, such that $b \leq T_B - T_A \leq B$. We assume that the plan contains no disjunctive bound-constraints between two events, i.e., the graph of events and bound-constraints is a Simple Temporal Network (STN) in the sense of Dechter, Meiri and Pearl [5]. Without loss of generality, we also assume that the STN graph is connected.

We concentrate on the process through which the executive selects individual events and executes them, i.e., assigns to them a specific time of occurrence that is consistent with the overall plan. It has been established [5] that finding the ranges of execution times for each event, the event's *time bounds*, is equivalent to solving two single-source shortest-path problems [4] on a simple transformation of the STN graph. Furthermore, if the STN is consistent, then for each event A it is possible to arbitrarily pick a time T_A within its time bounds and find corresponding times for the other events such that the set of occurrence times for all events satisfies the plan constraints. This suggests a "naive" execution algorithm that iteratively: (1) selects an event such that the current time is within the event's time bound and the event is *enabled*, i.e., all events that must directly precede it in the STN have been executed; (2) assigns the current time to the event; and (3) propagates the consequences of "collapsing" the event's time bounds to every other time bound. The iteration continues until all events have been executed.

There are two problems with the naive execution algorithm. The first is that precisely estimating the propagation cost for a general STN is difficult and may require considering the possible propagations in a large number of possible execution conditions. Without such analysis, the best we can do is to give a bound that depends on the total size of the plan; more precisely the bound corresponds to running the Dijkstra algorithm[1] twice on the graph. The complexity of this propagation is $O(e + n \log n)$, where e is the total number of edges and n the total number of nodes in the STN. A second, more serious problem is that selecting events on the sole basis of time bound information

[1]Since the STN is guaranteed to remain consistent, it is possible to avoid using the more costly Bellman-Ford-Moore algorithm.

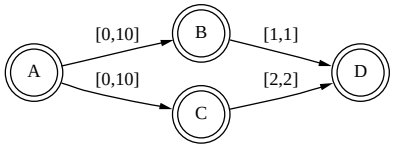

Figure 1: Simple Temporal Network.

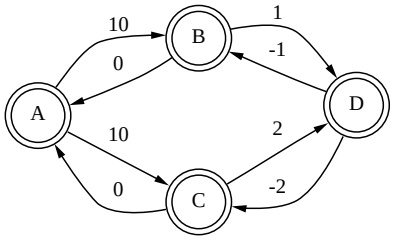

Figure 2: Distance Graph.

and precedence enablement may lead to incorrect executions. Consider the example network in Figure 1. Intuitively this network corresponds to two tasks BD and CD of fixed durations, respectively 1 and 2 time units, that synchronize at the end (event D) and must start within 10 time units of a time origin (event A). Figure 2 shows the corresponding *distance graph* [5], suitable for shortest-path propagation. If we assume that event A always occurs at time 0, events B and C will initially obtain time bounds $\langle 1, 10 \rangle$ and $\langle 0, 9 \rangle$ respectively. (Recall from [5] that the lower bound for a node is computed as the negation of the shortest-path distance from the node to the time origin, while the upper bound is simply the shortest-path distance from the origin to the node.) Suppose now that the current time is 5, but tasks BD and CD have not yet started. Since the time bounds of both B and C contain time 5, the naive execution algorithm may very well select for execution event B only to discover after propagation that event C should have started at time 4 in order for the plan to execute consistently. Thus, the naive approach does not guarantee correct execution of a plan under all execution conditions.

The problem with the plan in Figure 1 is that there is an implicit synchronization constraint that requires C to be executed exactly 1 time unit before B. When the execution reaches B and C, this implicit constraint can only be detected by increasing the lower bound of waiting events to the current time, and propagating, *before* considering which event to select. Although this fixes the consistency problem, it does not improve the real-time performance. Indeed, it makes it worse since

```
TIME DISPATCHING ALGORITHM:
    1. Let
         A = {start_time_point}
         current_time = 0
         S = {}
    2. Arbitrarily pick a time point TP in A such
       that current_time belongs to TP's time bound;
    3. Set TP's execution time to current_time and add
       TP to S;
    4. Propagate the time of execution
       to its IMMEDIATE NEIGHBORS in the distance
       graph;
    5. Put in A all time points TPx such that all
       negative edges starting from TPx have a
       destination that is already in S;
    6. Wait until current_time has advanced to
       some time between
          min{lower_bound(TP) : TP in A}
       and
          min{upper_bound(TP) : TP in A}
    7. Go to 2 until every time point is in S.
```

Figure 3: The Dispatching Execution Controller.

we may now have to propagate the new lower bound from several waiting events rather than from a single selected event.

On the other hand, to fix the performance problem, we would like to restrict the execution algorithm to use a *local propagation* that, on the basis of the execution time of an event, adjusts only the time bounds of the *neighboring* events. However, this makes the consistency problem worse. To see this, note that with the plan in Figure 1, even if C is executed first, local propagation would allow B to be executed *more* than one time unit after C, which also violates the implicit constraint.

Notwithstanding these considerations, there are many networks that *are* successfully executed by using the naive execution algorithm. This continues to be true even when the propagation is restricted to be local. In fact, as we will see, every consistent STN is equivalent to such a network.

Figure 3 shows a detailed local propagation algorithm that we call the *dispatching execution controller*. Note that the flexible wait in step 6 provides some room for responding to unmodeled external contingencies. This can include unexpected events (in contrast to work that deals with anticipated uncontrollable events [12]). Step 5 is a precise formulation of the *enablement* requirement that prevents execution of a node until all its enabling nodes have first been executed. Note that with this formulation a deadlock situation cannot occur, since in a consistent distance graph there are no negative cycles. An execution carried out by the dispatching execution controller is called a *dispatching execution*. An STN is said to be *dispatchable* if it is always correctly executed by the dispatching execution controller.

The propagation time needed to execute a dispatchable plan is easy to estimate, and varies directly with b, the maximum number of edges that can enter or exit an event in the associated distance graph.

In this paper, it is shown that every consistent STN can be reformulated as an equivalent dispatchable network. This is achieved by (1) constructing the all-pairs shortest path network (which is shown to be dispatchable), and (2) eliminating unneeded edges to obtain an equivalent dispatchable network of minimum size.

3 Finding dispatchable networks

We will use the following notation with respect to distance graphs. Given a timepoint X, the expression T_X denotes its execution time with respect to some schedule or execution. If X and Y are two timepoints, XY denotes an edge from X to Y, and $b(X, Y)$ is its distance or length. (The edge XY represents the constraint $T_Y - T_X \leq b(X, Y)$.) We write $|XY|$ to denote the distance along a shortest path from X to Y, or ∞ if no path exists. (Note that $|XY|$ may be negative in distance graphs.) The proofs of the theorems (and supporting lemmas) are contained in the Appendix.

The first result is useful for simplifying the local propagation required in a dispatching execution. Recall [5] that in an STN distance graph, the upper bounds of timepoints are propagated in the forward direction of edges, whereas lower bounds are transmitted in the reverse direction.

Theorem 1 *In a dispatching execution, upper-bound propagations along negative edges, and lower-bound propagations along non-negative edges, are both ineffectual, i.e., they do not affect the course of the execution.*

Note that theorem 1 shows that, in a dispatching execution, the upper and lower-bound propagations can be confined to disjoint sets of edges.

In the remainder of the paper, unless stated to the contrary, it is convenient to use the term execution to mean dispatching execution.

We next investigate what is needed to obtain dispatchable networks. Recall that any STN can be rewritten as an All-Pairs shortest-path network (called the *d-graph* in [5]). We have the following.

Theorem 2 *Every All-Pairs shortest-path network is dispatchable.*

Although the All-Pairs network is dispatchable, it has some obvious disadvantages. In particular, the prop-

agation at each node requires time proportional to n, the number of nodes, in *every* case. Fortunately, we can do better. Relying on Theorem 2, we will adopt the following strategy. Given an arbitrary STN, we first construct the equivalent All-Pairs network. Then we strip out unneeded edges, the goal being to end up with an equivalent dispatchable network of manageable size. Although it is possible to construct examples where there are no unneeded edges, experiments show that, typically, a minimal dispatchable network is found that is of size comparable to that of the original network.

To make this work, we need some means of identifying unneeded edges. Formally, an edge is *unneeded* if its removal does not admit any new executions. Intuitively, this condition is satisfied if its propagations are always superseded by those of some other edge. By Theorem 1, the only cases we need to consider are forward propagations along non-negative edges, which may affect upper-bounds, and backward propagations along negative edges, which may affect lower-bounds.

Recall that for an edge XY, the expression $T_X + b(X,Y)$ constitutes the upper-bound value propagated forward from X to Y, while $T_Y - b(X,Y)$ is the lower-bound value propagated backwards from Y to X.

This leads to the following definition.

Definition 1 *(a) Consider two edges AC and BC with the same destination C, and suppose the lengths of both are non-negative. We say BC* upper-dominates *AC if in every consistent execution, $T_B + b(B,C) \leq T_A + b(A,C)$.*

(b) Consider two edges AC and AB with the same source A, and suppose the lengths of both are negative. We say AB lower-dominates *AC if in every consistent execution, $T_B - b(A,B) \geq T_C - b(A,C)$*

(c) Finally, we say an edge $E1$ dominates *an edge $E2$ if either $E1$ upper-dominates $E2$ or $E1$ lower-dominates $E2$.*

The following results pertain to graphs that satisfy the triangle inequality. Note that these include the All-Pairs graph and subgraphs derived from it by removing edges.

The next theorem provides a characterization of the dominance relation that is more easily checked by an algorithm.

Theorem 3 (Triangle Rule) *Consider a consistent STN where the associated distance graph satisfies the triangle inequality.*

(1) A non-negative edge AC is upper-dominated by another non-negative edge BC if and only if $|AB| + |BC| = |AC|$.

(2) A negative edge AC is lower-dominated by another negative edge AB if and only if $|AB| + |BC| = |AC|$.

We next consider the removal of edges. We are interested in knowing whether this allows a new execution that *deviates* or differs from those that were possible before the removal. We will say an edge is *unneeded* if its removal does not produce a new dispatching execution. In this case, removing the edge will not introduce an incorrect execution that did not exist before.

The following result confirms our interest in the dominance relation.

Theorem 4 (Filtering Theorem) *An edge in a dispatchable network that satisfies the triangle inequality is unneeded if and only if it is dominated by some other edge.*

Theorems 3 and 4 together allow us to remove an edge AC from the All-Pairs shortest-path network if there is another node B such that $|AB| + |BC| = |AC|$, and either both $|AC|$ and $|AB|$ are negative, or both $|AC|$ and $|BC|$ are non-negative. In the former case, AC is lower-dominated by AB, while in the latter, it is upper-dominated by BC.

Notice that since the removal of a dominated edge leaves the set of executions unchanged, it does not interfere with the dominance relation between other pairs of edges. This suggests a potential for multiple removals, where the triangle rule conditions can conveniently be checked in the fixed All-Pairs network. However, some interaction is still possible where edges dominate each other; obviously, only one may be removed on account of the other (although they may both be removed if dominated by a third edge). To see how edge removals may be combined, we consider further properties of the dominance relation.

Theorem 5 *The dominance relation is reflexive and transitive.*

A binary relation that is reflexive and transitive is called a *preorder*. It is well-known that a preorder $\leq$ induces an equivalence relation $\equiv$, defined by $x \equiv y$ if $x \leq y$ and $y \leq x$. Moreover, the equivalence classes are partially ordered by the $\leq$ relation.

In the case of the dominance relation, the induced equivalence classes will be useful in formulating a multiple removal strategy, as discussed in the next section.

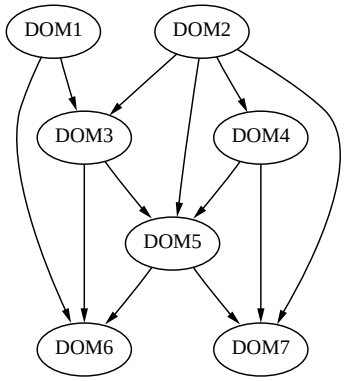

Figure 4: Domination Partial Order.

4 Minimality of filtered network

From the properties of the DOMINATES relations (Definition 1) we see that an all-pair shortest path network can give rise to a number of "minimal" dispatchable networks, where minimality means that the filtered network has a minimal number of edges. (Not to be confused with the "tightness" minimality property defined in [5].) We now wish to show that all of these have the same number of edges.

An example of a partial order structure induced by DOMINATES is shown in Figure 4.

Each set DOMi corresponds to an equivalence class for the DOMINATES relation. All links in such a class dominate each other symmetrically. The link from one equivalence class to another (e.g., from DOM3 and DOM5) represents the fact that any node in the first class dominates all of the nodes in the second class. This property follows straightforwardly from the transitivity of the DOMINATES relation. A minimal number of globally dominating edges can be obtained by picking one bound per each "source" DOMj equivalence class (in the case in Figure 4, DOM1 and DOM2). Since this selection can be done arbitrarily for each "source" DOMi, in general there is a potentially very large number of different minimal dispatchable networks obtainable from an all-pair shortest path network. However, from the point of view of the execution controller, all of the networks are equivalent, and they all have the same number of edges, so generating any one of them is sufficient.

5 Edge filtering algorithm

In this section we describe an algorithm that generates one of the minimal networks. First we describe the algorithm and then we prove its correctness and minimality.

```
procedure MARK-EDGES-FOR-ELIMINATION
begin
  for each pair of intersecting edges
  do begin
    if both dominate each other
    then
        if neither is marked
        then
            Pick one arbitrarily
            and mark it
        else
            Do nothing
    else if one dominates the other
    then
        Mark the dominated edge
  end
end
```

Figure 5: Minimal dispatch filtering algorithm.

Input: A consistent all-pair shortest path graph $< N, b(.,.) >$ where N is a set of time points with cardinality n and $b(.,.)$ is a total function $N \times N \rightarrow \Re$ such that $b(X, Y)$ is the length of the shortest path link from X to Y.

Output: A consistent minimal dispatchable network $< N, b'(.,.) >$, where $b'(.,.)$ is a restriction of $b(.,.)$ to a subset of $N \times N$.

The central routine in the algorithm is shown in figure 5. The routine visits edges in the network, marking some of them for elimination. A subsequent routine deletes the marked edges. The dominance relations can be established by applying the Triangle Rule of Theorem 3.

In the marking algorithm, two edges *intersect* if they either have the same source or the same destination. As a matter of implementation, all the pairs of intersecting edges can be conveniently visited by iterating over each set of three vertices or *triangle* and considering the edges between them.

It remains only to show that, with respect to the dominance partial order, as depicted in Figure 4, the application of the marking algorithm will mark all edges in the "non-source" equivalence classes (ones that have a predecessor class), and will eliminate all but one edge in the "source" equivalence classes.

First, consider an edge belonging to a "non-source" equivalence class DOMi. Eventually, it will be tested against an edge in an equivalence class DOMj that

precedes DOMi, and at that time it will be marked for elimination.

Next we treat the case of a "source" equivalence class DOMk. Consider the last time that the marking algorithm is applied to a pair of edges in DOMk that are both unmarked. Only one of the two edges will survive. The one that survives will survive until the end, since all other applications of the marking algorithm to pairs of edges in DOMk will necessarily involve at least one marked edge, which will prevent any additional marking from occurring. Furthermore, there must be exactly one edge left. Suppose to the contrary that there are at least two survivors E1 and E2 in DOMk. At some point the algorithm would have considered E1 and E2 as a pair, and if both were unmarked, it would have marked one of them. Thus, both could not have survived. It follows that only one edge per source equivalence class will survive after the termination of the algorithm.

Observe that when two unmarked edges dominate each other, there is a choice of which to eliminate. Thus, there are many equivalent minimal graphs that could be produced by the algorithm. Notice, however, they all contain exactly one edge from each of the "source" equivalence classes, and so they all have the same number of edges. This shows the algorithm is "best possible," in the sense that it produces a graph with a globally minimum number of edges.

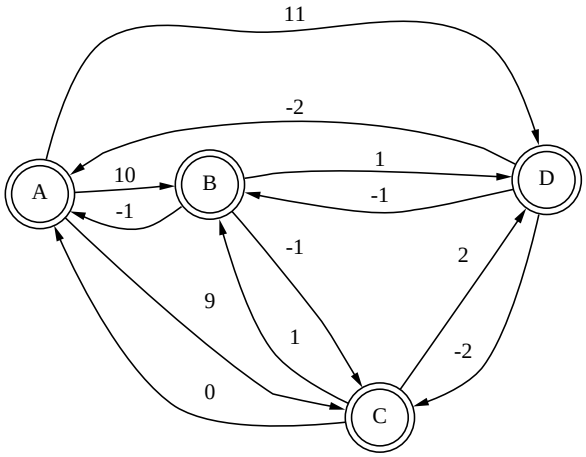

Figure 6: All Pairs Graph.

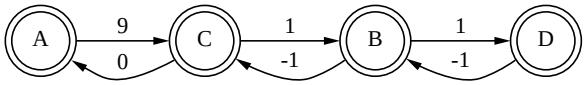

Figure 7: Final Filtered Graph.

6 Example and Experimental results

Continuing the example started with Figure 1, Figure 6 represents the fully connected distance graph obtained after application of the all-pairs shortest-path propagation. (Note that the BA and AC distances have decreased from the edge values in the original distance graph due to alternate, shorter paths.) After the application of the filtering algorithm described in section 5 we obtain the minimal dispatchable graph in figure 7. Notice that the final STN contains one less time-bound edge than the starting network in figure 1.

The algorithm described in Section 5 already has a practical application. It is being used in the New Millennium Remote Agent [10], a control architecture that will operate autonomously the Deep Space 1 (DS1) spacecraft in a 6 day experiment scheduled for October 1998. Table 1 summarizes the experimental results on the three plans that will be nominally generated and executed during the experiment.

All the results refer to distance graphs like those in figure 2, figure 6 and figure 7. The results show that the minimal dispatchable network is significantly smaller than the all-pairs dispatchable network, having between 5% and 7% the number of edges of the all-pairs network. They also show that the size of the minimal dispatchable network is smaller than the original plans generated by the on-board planner, having between 40% and 70% the edges of the original plan. Notice that even if the original plan were dispatchable, the minimal plans improve the real-time guarantee (proportional to the maximum number of branching edges at a node) between 1.5 and 2.2 times with respect to the original plan.

A Proofs

Lemma 1 *Given any consistent schedule for any STN, there is a dispatching execution that realizes the schedule.*

Proof: First we show that the enablement restriction does not exclude any consistent schedules. To see this, note that for any link $X \rightarrow Y$, we have $T_Y - T_X \leq b(X,Y)$, and so $T_Y < T_X$ if $b(X,Y)$ is negative. Second, note that the restriction to local propagation is actually more lenient in terms of nar-

Table 1: Minimal Dispatchability For DS1 Plans

	nodes	original edges	All-Pair edges	minimal edges	original max. branch	minimal max. branch
PLAN-1	56	390	3080	156	18	11
PLAN-2	39	144	1482	102	14	9
PLAN-3	66	424	4290	192	26	12

rowing the time bounds, so all consistent choices for execution time remain. □

In light of lemma 1, we may use the terms "consistent schedule" and "consistent execution" interchangeably in the subsequent proofs.

For the next result, recall that in STN propagation, the upper bounds of timepoints are propagated in the forward direction of edges, whereas lower bounds are transmitted in the reverse direction.

Theorem 1 *In a dispatching execution, upper-bound propagations along negative edges, and lower-bound propagations along non-negative edges, are both ineffectual, i.e., they do not affect the course of the execution.*

Proof: First we remark that a propagation to an already executed node is always ineffectual, since it cannot narrow the bounds further. Note also that such a propagation cannot generate an inconsistency, since the constraint has already been enforced by a prior propagation in the reverse direction. Now consider an upper-bound propagation along an edge $X \rightarrow Y$ that has a negative length. Because of the enablement condition, Y must have been executed before X. Thus, by the remark above, the propagation is ineffectual. Next consider a lower-bound propagation along a reverse edge $X \leftarrow Y$ that has non-negative length w. If Y has been executed before X, then we are done by the earlier remark. Otherwise, Y must occur at or after X. In that case, the edge constraint requires that $T_X - T_Y \leq w$, which can be rewritten as $T_Y \geq T_X - w$. Since Y is not occurring before X anyway, this bound does not constrain Y, and is subsumed by the actual execution time of Y. □

In the remainder of the proofs, it is convenient to use the term *execution* to mean *dispatching execution.*

Theorem 2 *Every All-Pairs shortest-path network is dispatchable.*

Proof: First we show that a full-propagating execution controller that respects the enablement conditions cannot generate an inconsistency. The theory of Simple Temporal Networks [5] guarantees that any locally consistent assignment can be extended to a global one. This means that (full) propagation during execution will not reduce any timepoint's bounds to the empty set. Thus, the only possibility for incorrect execution is if a pending unexecuted timepoint X is forced into the past by a propagation. For this to happen, there must be a shortest path of negative distance from some currently executing timepoint Y to X. In this case, because of the All-Pairs property, there will be a single edge from Y to X that has a negative length. But then the enablement condition would have forced X to be executed before Y, giving a contradiction.

Next, we observe that local propagation in the All-Pairs shortest-path network simulates full-propagation. It follows that the dispatching execution controller will not generate an inconsistency. Thus the All-Pairs network is dispatchable. □

Lemma 2 *Let A and B be timepoints in a consistent STN. Then in all consistent schedules, $T_B - T_A \leq |AB|$. Moreover, if $|AB|$ is finite, there is at least one consistent schedule where $T_B - T_A = |AB|$. If $|AB|$ is infinite, there are consistent schedules in which $T_B - T_A$ is arbitrarily large.*

Proof: The first part is immediate upon summing the inequalities for each edge in a shortest path. To see the second part, consider adding a link from B to A with length $-|AB|$. The network must still be consistent, since the shortest cycle through the edge BA has length $-|AB| + |AB| = 0$. Thus, there is at least one consistent schedule for the new network. This satisfies $T_A - T_B \leq -|AB|$ Combining this with the inequality of the first part gives $T_B - T_A = |AB|$. The result then follows, since this is also a consistent schedule for the original network. A similar method works for the infinite case by adding a negative link whose absolute value is arbitrarily large. □

Theorem 3 *Consider a consistent STN where the associated distance graph satisfies the triangle inequality.*

(1) A non-negative edge AC is upper-dominated by another non-negative edge BC if and only if $|AB| +$

$|BC| = |AC|$.

(2) A negative edge AC is lower-dominated by another negative edge AB if and only if $|AB| + |BC| = |AC|$.

Proof: First consider edges AC and BC in (1). Suppose $|AB|+|BC| = |AC|$. By the first part of lemma 2, in any consistent schedule, we have $T_B - T_A \leq |AB|$. It follows that $T_B + |BC| \leq T_A + |AB| + |BC|$. Thus, $T_B + |BC| \leq T_A + |AC|$ by our hypothesis. We can deduce from the triangle inequality that $|BC| = b(B,C)$ and $|AC| = b(A,C)$, so BC upper-dominates AC. Conversely, suppose that BC upper-dominates AC. Then in every consistent schedule, $T_B + |BC| \leq T_A + |AC|$. If $|AB| + |BC| \neq |AC|$, the only option allowed by the triangle inequality is $|AC| < |AB|+|BC|$. Combining the inequalities yields that $T_B - T_A < |AB|$ in every consistent schedule. But this contradicts the second part of lemma 2 if $|AB|$ is finite. In the infinite case, we can choose some finite value K such that $|AC| < K + |BC|$. Then $T_B - T_A < K$, which also contradicts lemma 2.

Now consider edges AC and AB in (2). Suppose again $|AB| + |BC| = |AC|$. The first part of lemma 2 gives us $T_C - T_B \leq |BC|$. Thus, $T_C + |AB| \leq T_B + |AB| + |BC| = T_B + |AC|$. This can be rewritten as $T_B - |AB| \geq T_C - |AC|$, and the result follows using the triangle inequality on $|AB|$ and $|AC|$. Conversely, if AB lower-dominates AC, then $T_B - |AB| \geq T_C - |AC|$ in every consistent schedule. Suppose $|AC| < |AB| + |BC|$. Then $T_C - T_B < C < |BC|$ for some C in every consistent schedule, again contradicting lemma 2. Thus, $|AC| = |AB| + |BC|$. □

For the following lemma, a *trace* of an execution or partial execution is a sequence $S_0, S_1, \ldots$ where S_i is the set of events that are executed at time i. A *deviation* occurs in a partial execution after an edge removal when the trace is no longer a prefix of any of the traces before the edge removal. This may happen either because of the addition or omission of an event in the trace.

Lemma 3 *Suppose the removal of an edge AC from a dispatchable network produces a new, deviant execution. If AC is a negative edge, then the earliest time at which the execution deviates is when A is executed. Otherwise it is when C fails to be executed within* $b(A,C)$ *time units after A. In either case, the AC constraint is violated.*

Proof: The following terminology will be useful. The *background state* of any time-point during an execution refers to its current time-bounds and enablement/execution status.

Consider first the case where AC is a negative edge. Then AC is an enablement edge for A. Let T be the time of first deviation in the schedule. Note that T cannot occur before A is executed, since before then the only differences in the background states of the timepoints are in the more lenient enablement status and lower bound of A, which only allow it to be executed sooner.

Now suppose the AC constraint is not violated. Then C must have been executed before A, and after C and A are executed, the background state of every timepoint will mirror that before the removal of AC. In that case, the execution will not deviate at or after the execution of A. Thus, the AC constraint must be violated. It follows that T cannot occur after A is executed, and so it must occur precisely when A is executed, and either C was not executed before A, or A occurs too soon after C.

The case where AC is non-negative is similar. Again let T be the time of first deviation. Note that before A is executed, the background state of every node (including A, since propagations from C to A are ineffectual) is the same as before the edge removal. Thus, T cannot occur before A is executed. After A is executed, the only difference in the background states is in the upper bound of C, which may be larger than before the edge removal. Again, if the AC constraint is satisfied, the execution will not deviate. Thus, the AC constraint must be violated, and the deviation occurs when C fails to execute within the allotted time after A. □

Theorem 4 *An edge in a dispatchable network that satisfies the triangle inequality is unneeded (in the sense that its removal does not alter the set of executions) if and only if it is dominated by some other edge.*

Proof: Suppose an edge AC is dominated by another edge. We will show the set of dispatching executions is unaltered after the dominated edge is removed. Consider first the easier case where a negative edge AC is lower-dominated by another negative edge AB. Suppose that after AC is removed, there is a new, deviant execution. By lemma 3, the deviation first occurs when A is executed. Note that B is still an enablement condition for A (AB was not removed), so B is executed before A. Since $T_B - b(A,B) \geq T_A - b(A,C)$, the propagation from B to A subsumes that from C to A by the time A is executed. This implies the execution time of A does not deviate, which is a contradiction.

Next consider the situation where a non-negative edge AC is upper-dominated by another non-negative edge

BC. Suppose that after AC is removed, there is a new, deviant execution. By lemma 3, AC is violated in the deviant execution, so A must be executed before C, and the deviation occurs $|AC|$ time units after A when C fails to be executed. (By the triangle inequality $b(A, C) = |AC|$.) By theorem 3, $|AC| = |AB| + |BC|$. Since $|BC| \geq 0$, it follows that $|AC| \geq |AB|$. Thus, B must be executed before the deviation occurs. Then by the dominance condition, the propagation from B to C subsumes that from A to C, implying that C does not deviate.

Conversely, suppose an edge AC is not dominated. Consider first the case in which AC is non-negative. By theorem 3, for any node B we have either BC is negative or $|AB| + |BC| > |AC|$. In the former case, any propagation from B to C is ineffectual by theorem 1. With respect to the latter cases, consider a propagation of distances bounds with A as the source. The upper bounds for each node will then form a consistent schedule. This shows that there is some execution in which $T_B = T_A + |AB|$ for every B such that $|AB| + |BC| > |AC|$. It follows that $T_B + |BC| > T_A + |AC|$ in this execution, for every such B. This means that if the edge AC is removed, the execution can be modified to delay the execution time of C until more than $|AC|$ time units after A, which is a deviation. Thus, AC is needed. The proof when AC is negative is similar. □

Theorem 5 *The dominance relation is reflexive and transitive.*

Proof: Reflexivity is immediate from the definitions. Transitivity follows from the fact that only dominances with edges of the same sign can be chained. □

Acknowledgments

We thank Pandu Nayak, David Smith, and the referees for suggestions to improve the presentation of these results.

References

[1] R. P. Bonasso, D. Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. *JETAI*, 9(1), 1997.

[2] J. Bresina, M. Drummond, , and S. Kedar. *Reactive, Integrated Systems Pose New Problems for Machine Learning.* Morgan Kaufmann, San Mateo, California, 1993.

[3] T. Carpenter, K. Driscoll, and K. Carciofini J. Hoyme. Arinc 659 scheduling: Problem definition. In *Proceedings of 1994 IEEE Real Time System Symposium.* IEEE, 1994.

[4] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT press, Cambridge, MA, 1990.

[5] R. Dechter, I Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, May 1991.

[6] Brian Drabble, Austin Tate, and Jeff Dalton. O-plan project evaluation experiments and results. Oplan Technical Report ARPA-RL/O-Plan/TR/23 Version 1, AIAI, July 1996.

[7] N. Muscettola, P. Morris, B. Pell, and B. Smith. Issues in temporal reasoning for autonomous control systems. In F. Anger, editor, *Working Notes from the 1997 AAAI workshop on Spatial and Temporal Reasoning*, 1997. available at http://ic-www.arc.nasa.gov/ic/projects/Executive/papers/aaai97-temporal.ps.

[8] N. Muscettola and B. Pell. Real-time execution of temporal plans. Technical Report in preparation, Computational Sciences Division, NASA Ames Research, 1997.

[9] David Musliner, Ed Durfee, and Kang Shin. Circa: A cooperative, intelligent, real-time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6), 1993.

[10] B. Pell, D. E. Bernard, S.A. Chien, E. Gat, N. Muscettola, P. P. Nayak, M.D. Wagner, and B.C. Williams. An autonomous spacecraft agent prototype. *Autonomous Robotics*, forthcoming, 1997.

[11] Reid Simmons. An architecture for coordinating planning, sensing, and action. In *Procs. DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 292–297, San Mateo, CA, 1990. DARPA, Morgan Kaufmann.

[12] T. Vidal and M. Ghallab. Dealing with uncertain durations in temporal constraint networks dedicated to planning. In *Proc. of 12th European Conference on Artificial Intelligence (ECAI-96)*, pages 48–52, 1996.

[13] D. E. Wilkins, K. L. Myers, J. D. Lowrance, and L. P. Wesley. Planning and reacting in uncertain and dynamic environments. *JETAI*, 7(1):197–227, 1995.