Incremental Dynamic Controllability in Cubic Worst-Case Time

Mikael Nilsson Jonas Kvarnström Patrick Doherty
Department of Computer and Information Science
Linköping University, SE-58183 Linköping, Sweden
{mikni,jonkv,patdo}@ida.liu.se

***Abstract*—It is generally hard to predict the exact duration of an action. Uncertainty in durations is often modeled in temporal planning by the use of upper bounds on durations, with the assumption that if an action happens to be executed more quickly, the plan will still succeed. However, this assumption is often false: If we finish cooking too early, the dinner will be cold before everyone is ready to eat. Simple Temporal Problems with Uncertainty (STPUs) allow us to model such situations. An STPU-based planner must verify that the plans it generates are executable, captured by the property of dynamic controllability. The EfficientIDC (EIDC) algorithm can do this incrementally during planning, with an amortized complexity per step of $O(n^3)$ but a worst-case complexity per step of $O(n^4)$. In this paper we show that the worst-case run-time of EIDC does occur, leading to repeated reprocessing of nodes in the STPU while verifying the dynamic controllability property. We present a new version of the algorithm, EIDC2, which through optimal ordering of nodes avoids the need for reprocessing. This gives EIDC2 a strictly lower worst-case run-time, making it the fastest known algorithm for incrementally verifying dynamic controllability of STPUs.**

I. Introduction and Background

When planning for multiple agents, for example a joint UAV rescue operation, generating concurrent plans is usually essential. This requires a temporal formalism allowing the planner to reason about the possible times at which plan events will occur during execution. A variety of such formalisms exist in the literature. For example, Simple Temporal Problems (STPs [1]) allow us to define a set of events related by binary temporal constraints. The beginning and end of each action can then be modeled as an event, and the interval of possible durations for each action can be modeled as a constraint related to the action's start and end event: $dur = end - start \in [min, max]$.

However, an STP solution is defined as *any* assignment of timepoints to events that satisfies the associated constraints. When an action has a duration $dur \in [min, max]$, it is sufficient that the remaining constraints can be satisfied for *some* duration within this interval. This corresponds to the case where the planner can freely *choose* action durations within given bounds, which is generally unrealistic. For example, nature can affect action durations: Timings of UAV flights and interactions with ground objects will be affected by weather and wind.

A formalism allowing us to model durations that we cannot directly control is STPs with Uncertainty (STPUs) [2]. This formalism introduces *contingent* constraints, where the time between two events is assumed to be assigned by nature. In essence, if an action is specified to have a contingent duration $d \in [t_1, t_2]$, the other constraints must be satisfiable for *every* duration that nature might assign within the given interval.

In general, STPUs cannot be expected to have static solutions where actions are scheduled at static times in advance. Instead we need dynamic solutions with a mechanism for taking into account the observed times of uncontrollable events (the observed durations of actions). If such a dynamic solution can be found, the STPU is *dynamically controllable* (DC) and the plan it represents can be executed regardless of the outcomes of the contingent constraints.

Planning with STPUs. Many automated planners begin with an empty plan and then incrementally add one new action at a time using some search mechanism such as forward search or partial-order planning. The initial empty plan is trivially dynamically controllable. If we add an action to a DC plan, the result may or may not be DC. On the other hand, the DC property is monotonic in the sense that if we add an action or a new constraint to a *non-DC* plan, the result is guaranteed *not* to be dynamically controllable. Thus, if the planner generates a non-DC plan at some point during search, extending the plan is pointless. In this situation the search tree can be pruned.

The earlier this opportunity for pruning can be detected, the better. Ideally, the planner should determine after each individual action is added whether the plan remains DC. Dynamic controllability will then be verified a large number of times during the planning process, necessitating a fast verification algorithm. For most of the published algorithms, this would require (re-)testing the entire plan in each step [3]–[6]. This takes non-trivial time, and one can benefit greatly from using an *incremental* algorithm instead. The fastest known such algorithm at the moment is the *EfficientIDC* (EIDC) algorithm [7]. The EIDC algorithm has a worst-case run-time in $O(n^4)$ and an amortized run-time in $O(n^3)$.

In this paper we show that the worst-case run-time of EIDC can occur: There are cases where the number of steps required after the addition of a single action is proportional to n^4. The higher complexity when processing such STNUs is caused by EIDC having to reprocess nodes repeatedly since it processes them in a non-optimal order. We create a new version of EIDC, EIDC2, which overcomes these problems by determining a better order in which nodes can be processed. This gives EIDC2 a worst-case run-time in $O(n^3)$ without amortization.

II. Definitions

We now define the fundamental concepts used in this paper.

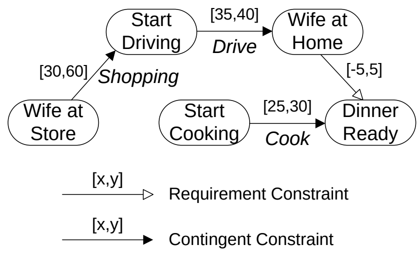

Fig. 1. Example STNU.

Definition 1: A **simple temporal problem** (**STP**) [1] consists of a number of real variables $x_1,\ldots,x_n$ and constraints $T_{ij} = [a_{ij}, b_{ij}], i \neq j$ limiting the distance $a_{ij} \leq x_j - x_i \leq b_{ij}$ between the variables. □

Definition 2: A ***simple temporal problem with uncertainty*** (**STPU**) [2] consists of a number of real variables $x_1,\ldots,x_n$, divided into two disjoint sets of *controlled timepoints* R and *contingent timepoints* C. An STPU also contains a number of *requirement constraints* $R_{ij} = [a_{ij}, b_{ij}]$ limiting the distance $a_{ij} \leq x_j - x_i \leq b_{ij}$, and a number of *contingent constraints* $C_{ij} = [c_{ij}, d_{ij}]$ limiting the distance $c_{ij} \leq x_j - x_i \leq d_{ij}$. For the constraints C_{ij} we require $x_j \in C$ and $0 < c_{ij} < d_{ij} < \infty$. □

We will work with STPs and STPUs in graph form, with timepoints represented as nodes and constraints as labeled edges. They are then referred to as Simple Temporal Networks (STNs) and STNs with Uncertainty (STNUs), respectively. An example is shown in Figure 1. In this example, a man wants to cook for his wife. He does not want her to wait too long after she returns home, nor does he want the food to wait too long. These two requirements are captured by a single requirement constraint, whereas the uncontrollable (but bounded) durations of shopping, driving home and cooking are captured by the contingent constraints. The question is whether this can be guaranteed regardless of the outcomes of the uncontrollable durations, assuming that these are observable.

Definition 3: A **dynamic execution strategy** [3] is a strategy for assigning timepoints to controllable events during execution, given that at each timepoint, it is known which contingent events have already occurred. The strategy must ensure that all requirement constraints will be respected regardless of the outcomes for the contingent timepoints.

An STNU is **dynamically controllable (DC)** [3] if there exists a dynamic execution strategy for executing it. □

In Figure 1 a dynamic execution strategy is to start cooking 10 time units after receiving a call that the wife starts driving home. This guarantees that cooking is done within the required time, since she will arrive at home 35 to 40 time units after starting to drive and the dinner will be ready within 35 to 40 time units after she started driving. Note that there exists no valid static execution strategy where it is determined in advance when cooking should start.

Any STN can be represented as an equivalent *distance graph* [1]. Each constraint [u,v] on an edge AB in an STN is represented as two corresponding edges in its distance graph: AB with weight v and BA with weight $-u$. The weight of an edge XY then always represents an upper bound on the temporal distance from its source to its target: $time(Y) - time(X) \leq weight(XY)$. Computing the all-pairs-shortest-path (APSP) distances in the distance graph yields a *minimal representation* containing the tightest distance constraints that are implicit in the STN [1]. This directly corresponds to the tightest interval constraints $[u', v']$ implicit in the STN. If there is a negative cycle in the distance graph, then no assignment of timepoints to variables satisfies the STN: It is *inconsistent*.

Similarly, an STNU always has an equivalent *extended distance graph* (EDG) [6]. The example graphs in this paper are all STNUs in EDG form with the exception of Figure 1.

Definition 4: An **extended distance graph** (**EDG**) is a directed multi-graph with weighted edges of five kinds: positive requirement, negative requirement, positive contingent, negative contingent and conditional. □

Requirement edges and contingent edges in an STNU are translated into pairs of edges of the corresponding type in a manner similar to what was described for STNs. A conditional edge [6] is never present in the initial EDG corresponding to an STNU, but can be derived from other constraints through calculations discussed in the following sections.

Definition 5: A **conditional edge** CA annotated $< B, -w >$ encodes a conditional constraint: C must execute either after B *or* at least w time units after A. The node B is called the *conditioning node* of the constraint/edge. The edge is *conditioned on* the node B. □

III. DC VERIFICATION TECHNIQUES

Morris, Muscettola and Vidal [3] were the first to present a way of efficiently (see [8]) verifying if an STNU is dynamically controllable. Their algorithm made use of STNU-specific tightening rules, also called derivation rules. Each rule could be applied to a triangle of nodes and if certain conditions were met, new previously implicit constraints were derived and added explicitly to the STNU. They showed that if these derivation rules were applied to an STNU until quiescence (no rule application can generate new conclusions), all constraints needed for DC verification would be derived. The original algorithm made intermediate checks while adding constraints to make sure that conditions required for DC were still valid.

The derivation rules of Morris et al. provides a common ancestor theory for all non-TGA-based (see related work) DC verification algorithms. The original semantics were later revised [9] and the derivations refined [6]. However, the idea of deriving constraints from triangles of nodes is still valid.

Incremental dynamic controllability verification verifies the state of an STNU as DC or non-DC given a list of incremental changes. The list could contain addition of new nodes/edges or tightenings of already existing constraints. For simplicity, in the rest of this paper we will assume that only one change is made to the STNU before calling an incremental algorithm.

IV. FASTIDC

The FastIDC algorithm [10] was the original incremental DC verification algorithm. It introduced structure to derivations using so called *focus edges*. Instead of treating a triangle as an entity for derivation it focused on what could be derived

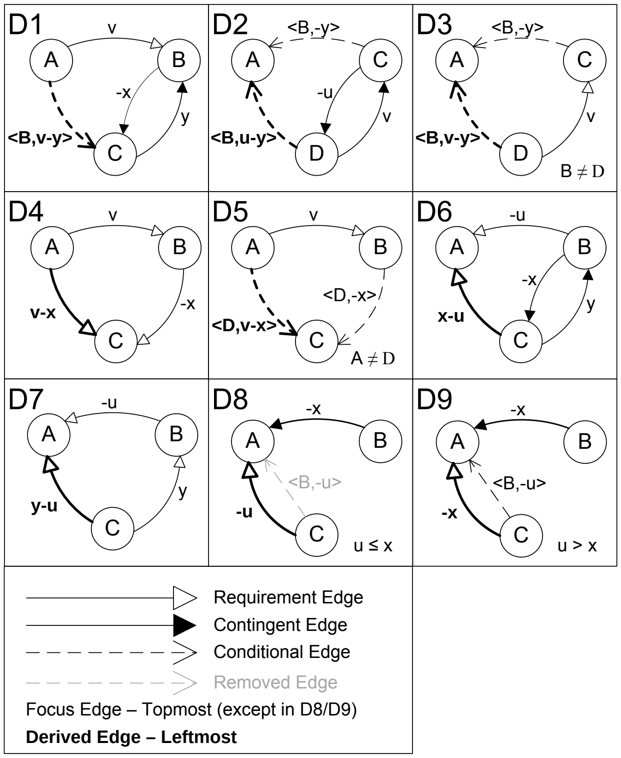

Fig. 2. FastIDC derivation rules.

from a *single* edge. Figure 2 shows FastIDC's derivation rules. Since an edge is responsible for the derivation of another edge, derivations by FastIDC can be seen as forming chains.

The original version of FastIDC was not correct. Some additions had to be made to capture all cases where the STNU were non-DC. We will use the modified, corrected version [11].

As this algorithm is correct it will be used as our basis for proving correctness of EIDC2. We will not give the details of FastIDC here. For correctness it does not matter in which order the derivations are applied. Only the fact that they are applied until quiescence is reached and that the algorithm detects negative cycles and squeezes. A *squeeze* [3] is when tighter bounds are imposed on an existing contingent constraint. If this happens, then there are cases where not all potential outcomes of uncontrollable durations are permitted, and the corresponding STNU is not DC. Similarly an STNU containing a negative cycle cannot be DC since a negative cycle corresponds to an impossible-to-satisfy constraint.

V. The Original *EfficientIDC* Algorithm

FastIDC may derive edges between the same nodes several times, which is problematic for its performance. Even though this may be prevented to a certain degree [7], it remains a problem to efficiently handle derivations in regions containing many unordered nodes, i.e. nodes that are mostly connected by positive requirement edges.

To overcome these problems a new algorithm was proposed [7]: The Efficient Incremental Dynamic Controllability checking algorithm (Algorithm 1, *EfficientIDC* or *EIDC* for short). EIDC uses focus *nodes* instead of focus edges to gain efficiency. It is based on the same derivations as FastIDC (Figure 2) but applies them in a different way. When EIDC updates an edge in the EFG, the target of the edge, and the source in some cases, are added to a list of focus nodes to be processed. When EIDC processes a focus node n, it applies all derivation rules that have an incoming edge to n as focus edge. This guarantees that no tightenings are missed [7].

The use of focus nodes allows EIDC, with the help of a modified version of Dijkstra's algorithm, to efficiently process parts of an EDG in a way that avoids certain forms of repetitive intermediate edge tightenings that are performed by FastIDC. The key to understanding this is that derivation rules essentially calculate a certain form of shortest distances in an EDG. For example, rule D4 states that if edge *AB* is tightened and there is an edge *BC*, the edge *AC* may have to be tightened to indicate the length of the shortest path between *A* and *C*. Shortest path algorithms cannot be applied indiscriminately, since there are complex interactions between the different kinds of edges, but can still be applied to subsets of an EDG in important cases.

We will now briefly discuss how EIDC works, both to provide a basis for the analysis showing why the algorithm can sometimes require $\Omega(n^4)$ time and to prepare for EIDC2.

Additional Data Structures. EIDC makes use of two additional separate data structures for efficiency.

The *Cycle Checking Graph* (*CCGraph*) has the same nodes as the EDG and has one unlabeled edge for every negative edge in the EDG. This is used to efficiently detect negative cycles using an incremental topological ordering algorithm [11].

The *Dijkstra Distance Graph* (*DDG*) is used to execute the modified version of Dijkstra's single source shortest path algorithm [7]. It contains the following information:

1) The positive requirement edges of the EDG, but placed in the opposite direction.
2) The negative contingent edges of the EDG, with weights replaced by their absolute values.

Updates to the DDG are trivial and are omitted for readability.

The EfficientIDC algorithm. As shown in Algorithm 1, EIDC has four parameters. In the first call, the extended distance graph G will contain two nodes and a single edge; in subsequent calls it consists of the output of the previous call to EIDC where exactly one edge has been added or tightened, and one node may have been added. The DDG D and CCGraph C are the *Dijkstra Distance Graph* and the Cycle Checking Graphs as explained above. These both persist through subsequent calls to the algorithm. Finally, the edge e is the newly added or tightened edge.

First the target of e is put in todo. If e is a *negative* requirement edge with no corresponding edge in the *CCGraph* C, it essentially introduces a new forced temporal ordering between two nodes. A corresponding edge is then added to the *CCGraph C*. If this causes a negative cycle, G is inconsistent and consequently not DC. Else, the source of e is added to *todo*: Processing this node may result in new edges into *Target*(e), and for efficiency this should preferably be done as early as possible (more on this later).

Iteration. As long as there are nodes in *todo*, a node to

Algorithm 1 The EfficientIDC Algorithm

```
function
EFFICIENTIDC(EDG G, DDG D, CCGraph C, edge e)
    todo ← {Target(e)}
    if e is negative and e ∉ C then
        add e to C
        if negative cycle detected then
            return false
        end if
        todo ← todo ∪ {Source(e)}
    end if
    while todo ≠ ∅ do
        current ← pop some n from todo where
            ∀e ∈ Incoming(C,n) : Source(e) ∉ todo
        PROCESSCOND(G,D,current)
        PROCESSNEGREQ(G,D,current)
        PROCESSPOSREQ(G,current)
        for each edge e added to G in this iteration do
            if Target(e) ≠ current then
                todo ← todo ∪ {Target(e)}
            end if
            if e is a negative req. edge and e ∉ C then
                add e to C
                if negative cycle detected then
                    return false
                end if
                todo ← todo ∪{Target(e),Source(e)}
            end if
        end for
        if G is squeezed then return false
        end if
    end while
    return true
end function
```

Algorithm 2 Helper Functions

```
function PROCESSCOND(G,D,current)
    allcond ← IncomingCond(current,G)
    condnodes ← {n ∈ G | n is the conditioning node of
            some e ∈ allcond}
    for each c ∈ condnodes do
        edges ← {e ∈ allcond | conditioning node of e is c}
        minw ← |min{weight(e) : e ∈ edges)}|
        add minw to the weight of all e ∈ edges
        for e ∈ edges do
            add e to D with reversed direction
        end for
        Dijkstra(current, D, minw)
        for all nodes n reached by Dijkstra do
            newW ← Dist(current,n) − minw
            e ← cond. edge (n → current), weight newW
            if e is a tightening then
                add e to G
                apply D8 and D9 to e
            end if
        end for
        Revert all changes to D
    end for
    return
end function
function PROCESSNEGREQ(G,D,current)
    edges ← IncomingNegReq(current,G)
    minw ← |min{weight(e) : e ∈ edges)}|
    add minw to the weight of all e ∈ edges
    for e ∈ edges do
        add e to D with reversed direction
    end for
    Dijkstra(current, D, minw)
    for all nodes n reached by Dijkstra do
        newW ← Dist(n) − minw
        e ← req. edge (n → current) of weight newW
        if e is a tightening then
            add e to G
        end if
    end for
    Revert all changes to D
    return
end function
function PROCESSPOSREQ(G,current)
    for each e ∈ IncomingPosReq(current,G) do
        apply derivations D1, D4-D5 with e as focus edge
        if e is conditional edge then
            apply derivations D8-D9 with e as focus edge
        end if
    end for
    return
end function
```

process, *current*, is selected and removed from the set. Incoming negative edges e to the chosen node *current* must not originate in a node also marked as *todo*: Otherwise, $Source(e)$ should be processed first, since this has the potential of adding new incoming edges to *current*. There is always a *todo* node satisfying this criterion, or there would be a cycle of negative edges that would have been detected.

The incoming edges to the selected node are processed in three steps with the help of three helper-functions, shown in Algorithm 2. For a given node N, PROCESSCOND() generates all of the conditional edges with destination N, while also possibly generating some requirement edges (positive or negative) with destination N. PROCESSNEGREQ() generates all of the negative requirement edges with destination N, while also possibly generating some positive requirement edges. Finally, PROCESSPOSREQ() uses the positive requirement edges with destination N to generate additional edges with other destinations. In this way, the helper functions, when executed in the given order, effectively generate all edges with destination N.

We will describe the functions in reverse order to allow the introduction of EIDC concepts gradually.

Incoming positive requirement edges are processed using PROCESSPOSREQ(). This simply applies rules D1, D4-D5 and D8-D9 to these edges, which are exactly those rules that FastIDC could apply with such edges as focus.

Incoming negative requirement edges are processed using PROCESSNEGREQ(). This function is equivalent to applying rules D6 and D7, which are exactly those rules FastIDC could apply with a negative requirement edge as focus. However, the function does this for a larger part of the graph in a single step.

Rule D7 combines a sequence of a positive requirement edge *CB* (weight *y*) and a negative requirement edge *BA* (weight $-u$) and generates an edge *CA* of weight $y-u$, unless an edge of lower weight is already present. This is essentially one step in a shortest path calculation, which in FastIDC can "spread out" from node $A = current$ in a large number of iterations that incrementally generate shorter and shorter paths to the same nodes. (D6 will be considered later.)

Since $A = current$ (the target of the focus edge), we are calculating the shortest paths to a certain *target* (*current*). This can also be achieved by calculating the shortest path from a certain *source* in a graph containing all relevant edges with reversed directions. For D7 the relevant edges are the positive requirement edges, which are always present in reversed form in the DDG, and the negative requirement edges leading to *current*, which are added in reversed form to the DDG by PROCESSNEGREQ().

A DDG constructed in this way would contain negative edges, requiring the use of an algorithm such as Bellman-Ford to generate shortest paths. However, all negative edges in the DDG originate in *current*, and there are no negative cycles. Therefore every shortest path from *current* must contain exactly one negative edge. We can therefore calculate shortest paths using Dijkstra's algorithm by determining the lowest negative edge weight *minw*, adding the absolute value of this to all negative edges, calculating shortest paths, and subtracting this from the final path costs. This will in a single call to Dijkstra derive a final set of shortest distances that FastIDC might have had to perform a large number of iterations to converge towards. It is this part of EIDC that allows efficient derivation over regions of unordered nodes.

Also, rule D7 is only triggered by *negative* requirement edges, and can therefore only trigger another instance of itself for as long as it generates *new* negative edges. We therefore do not have to generate paths of weight greater than zero. In the DDG where *minw* has been added to every path weight, we can stop searching when reaching a path length of *minw*.

Rule D6 is handled in a similar manner, with minor differences. First, D6 combines a negative requirement focus edge *BA* of weight $-u$ with a *negative contingent* edge *BC* of weight $-x$ to yield a new edge. These negative contingent edges already point "away" from the current node $A = current$ and do not have to be reversed. However, their *signs* are inversed in the weight calculation in D6: The weight of the new edge should be $x-u$, not the weight sum $-x-u$. Therefore the signs are also inverted in the DDG, as stated earlier.

After running Dijkstra's algorithm, we know it is possible to derive a non-negative requirement edge to every node that was reached. We iterate over those nodes, determine the appropriate weight for each edge, and adds the edge if a tighter or equivalent edge did not already exist. Finally, the temporary changes to D made by this procedure are rolled back.

Note that any new incoming *negative* edge that is added to the EDG by PROCESSNEGREQ() is automatically processed by this same call to PROCESSNEGREQ() and therefore does not require further processing.

Incoming conditional edges are processed in a similar way in PROCESSCOND(). This function is equivalent to applying rules D2, D3, D8 and D9, which are exactly those rules FastIDC could apply with a conditional edge as focus (while D1/D5 also involve conditional edges, they have requirement edges as focus). However, as above, the function does this for a larger part of the graph in a single step.

PROCESSCOND() processes edges for the same conditioning node together, but edges for different conditioning nodes separately. This is possible because edges with different conditioning nodes are "independent": When the relevant derivation rules (D2, D3, D8, D9) are applied to an edge with conditioning node *n*, the result is either a requirement edge or a conditional edge with the *same* conditioning node *n*.

Thus, for every conditioning node *c* used by the incoming conditional edges, PROCESSCOND() first finds all edges *allcond* that are conditioned on *c* and have *current* as target. These are the edges that could be the focus of rules D2/D3. Here, D2 directly corresponds to D6 in PROCESSNEGREQ() above, and D3 directly corresponds to D7: Weight calculations are identical, the only difference being that both the focus edge and the tightened edge are conditional edges, not requirement edges. This difference is reflected in the fact that we *consider* incoming conditional edges and *create* new conditional edges.

Rules D8 and D9 applied once for each new conditional edge derived by Dijkstra. Derivation through D8 may add a further edge to *G*.

After processing incoming edges, EIDC checks all new edges that were derived by the helper functions. Edges that do not have *current* as a target need to be processed, so their targets are added to *todo*. If there is a negative requirement edge that is not already in the *CCGraph*, this edge represents a new forced ordering between two nodes. It must be added to the *CCGraph*, which is then checked for negative cycles. If a new edge is added to the *CCGraph* both the source and the target of the edge must be added to *todo* for efficiency as mentioned before. Finally, EIDC checks whether some edge in the EDG is squeezed at which point the STNU is known to be not DC.

See [7] for an extensive example processed by EIDC.

VI. ANALYSIS OF EIDC'S WORST-CASE SCENARIO

In this section we prove that EIDC's worst-case complexity of $O(n^4)$ is not an overestimate: Scenarios in which EIDC has to reprocess $\Omega(n)$ nodes do exist. We will only show a small subset of an STNU in which EIDC reprocesses one node. Figure 3 shows the starting situation where the EDG is DC and there is no more possible derivation to be found by EIDC. For simplicity we have only included the edges leading to reprocessing. The *X*, *Y* and *Z* nodes are special and will be discussed later. The other nodes are named alphabetically in the order they are processed by EIDC, starting with *a*. For the remainder of this example all edges we consider are requirement edges. A dashed pattern is used for negative edges whereas the positive edges have a continuous stroke. We use a bold font for nodes that are in *todo* and a gray font for nodes that are processed. We note that *Y* has many incoming positive requirement edges from some other part of the EDG.

Each external modification (each new or tightened edge resulting in a call to EIDC) will be called an *increment*. Table I

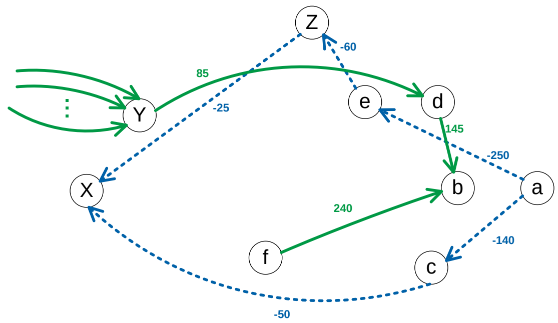

Fig. 3. Starting scenario.

TABLE I. Edges derived in the example.

Step	Chosen node	Added edges	*todo*
0			$\{a\}$
1	a	$b \to c$, $b \to e$	$\{b,c,e\}$
2	b	$d \to c$, $d \to e$, $f \to c$, $f \to e$	$\{c,e,d\}$
3	c	$f \to X$, $d \to X$	$\{e,d,X\}$
4	X	no new edge added	$\{e,d\}$
5	d	$Y \to e$	$\{e\}$
6	e	$f \to Z$, $Y \to Z$	$\{Z\}$
7	Z	$f \to X$, $Y \to X$	$\{X,Y\}$

shows what happens when EIDC processes the increment created by adding the $b \to a$ edge of weight 100. Initially, a is the only node in *todo*. In iteration 1, a must be chosen as *current* and processed, resulting in two new edges $b \to c$ and $b \to e$, and three nodes in *todo*. In iteration 2, b is the only node that can be chosen, because both c and e have negative incoming edges from b which is in *todo*. Processing b results in four new edges but still three nodes in *todo*, and so on.

Figure 4 shows the situation in step 4, when X is chosen for processing. The choice is in line with what the algorithm might do since it has no information about which choice is better. Figure 5 shows where we stop the example. At this point we see that X will again be added to *todo*. This is a problem leading to reprocessing and it is not the only one. As we can see we did process X before Y, so even if we did not get X in *todo* now, it will end up there again when Y is eventually processed. The first time X was processed all positive edges targeting Y were missed. In conclusion, the example shows multiple ways in which X will be chosen for reprocessing. A larger scenario may contain sections that are similar to the one in the example. Each of these will then lead to the reprocessing of one node. Since there are only 9 nodes in the example there is room for $\Omega(n)$ parallel subgraphs of similar size, all leading to the reprocessing of X. In the worst case X is the target of conditional edges conditioned on $\Omega(n)$ different uncontrollable nodes. This means that processing X takes $\Omega(n^3)$ time [7] for each reprocessing giving a total of time $\Omega(n^4)$ to process this increment. Therefore, the worst case stated in the analysis of EIDC complexity [7] is attained.

VII. EfficientIDC2

We will now propose a set of changes to EIDC to ensure that no nodes will have to be reprocessed within a given increment. The key behind the modification is a new way of choosing *current* nodes, where under certain circumstances

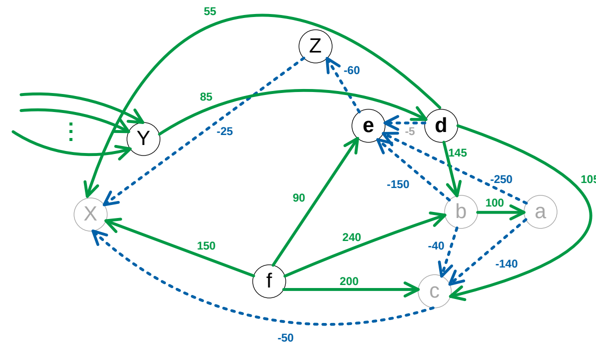

Fig. 4. The EDG at the time X is chosen for processing.

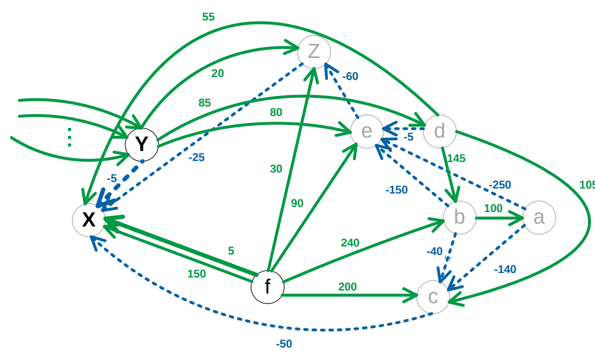

Fig. 5. Final part of the example.

additional nodes are available as candidates because of their potential to generate new incoming edges to the nodes that we already know need to be processed. Listing 3 shows the new algorithm, *EfficientIDC2* (*EIDC2* for short).

In contrast to EIDC, all nodes are added to *unprocessed* at the start since they may all need to be processed at some point. Note that since no nodes need to be reprocessed, no nodes re-enter *unprocessed* at a later stage. Only nodes that have modified incoming positive/negative edges or outgoing negative edges can be involved in the derivation of new edges. The algorithm puts all such nodes in the *active* set to keep track of nodes that must be processed at some point.

Therefore, when popping nodes from *unprocessed*, precedence is given to nodes in *active*. At some point the algorithm may reach a point where all nodes in *active* have been processed. Then the intersection between the sets is empty and the algorithm will halt. This happens at the point where EIDC would stop since no more nodes need to be processed, i.e. the *todo* set of EIDC became empty. In the next section we continue to look at the difference between the algorithms.

A. Comparison of Processed Nodes

As we have seen, EIDC processes only nodes that can lead to derivation of new edges (these are the *active* nodes of EIDC2). EIDC2 also processes these nodes, but to make sure that the nodes are processed in the correct order, EIDC2 will process any node that is executed after (as determined by following negative edges) any of the *active* nodes. This means that whereas EIDC2 avoids reprocessing it may in some

Algorithm 3 The EfficientIDC2 Algorithm

```
function EFFICIENTIDC2(EDG G, DDG D, CCGraph C, edge e)
    unprocessed ← {nodes ∈ G}
    active ← {Target(e)}
    if e is negative then
        active ← active ∪ {Source(e)}
        if e ∉ C then
            add e to C
            if negative cycle detected then
                return false
            end if
        end if
    end if

    while unprocessed ∩ active ≠ ∅ do
        current ← pop some n from unprocessed where
            ∀e ∈ Incoming(C,n) : Source(e) ∉ unprocessed,
            prioritizing popping nodes also in active
        PROCESSCOND(G,D,current)
        PROCESSNEGREQ(G,D,current)
        PROCESSPOSREQ(G,current)
        for each edge e added to G in this iteration do
            active ← active ∪ {Target(e)}
            if e is negative then
                active ← active ∪ {Source(e)}
                if e is a requirement edge and e ∉ C then
                    add e to C
                    if negative cycle detected then
                        return false
                    end if
                end if
            end if
        end for
        if G is squeezed then
            return false
        end if
    end while
    return true
end function
```

cases process more distinct nodes in total. In an STNU that is built incrementally where most constraints are added towards the end of the STNU, few nodes beside those needed by the increment will be processed to ensure optimal ordering.

B. Correctness

There has only been a sketched proof of correctness for EIDC. We will supply a more detailed proof here for EIDC2, not relying on the sketch for EIDC. This proof goes directly to the source, i.e. the derivations done by the corrected version of FastIDC [11]. A proof of correctness for the corrected version of FastIDC exists [8]. We divide this section into two parts: soundness followed by completeness. If the algorithm does not derive unsound constraints, and it derives all constraints needed for DC verification while also performing the same tests to detect non-DC properties, it is correct.

Soundness of EIDC2. The only edges generated by EIDC2 are generated by the sound derivation rules used by FastIDC and EIDC. Thus, EIDC2 is sound.

Completeness of EIDC2. Since FastIDC is complete, it suffices to show that whenever FastIDC derives a constraint, EIDC2 derives the same constraint.

We prove completeness in several steps and lemmas. In the following we will compare derivations of FastIDC and EIDC2. We assume that the algorithms have produced identical EDGs up until the current increment. Note that it does not matter for completeness in what order FastIDC applies its derivations, as long as they are applied whenever they can be, completeness follows. FastIDC may overwrite derived edges with tighter derivations at a later stage. We are only interested in the edges of the final EDG produced when running FastIDC.

We start with some notation: We will call the final EDG of FastIDC F and the final EDG of EIDC2 E. Corresponding nodes in the two EDGs will have the same name. To improve readability we will represent an edge from a to b by (a,b).

$after(X) = \{$nodes $n|$ there is a path of negative edges from n to X in $F\}$.

The definition means that $after(X)$ contains all the nodes that should be ordered after X when the increment is fully processed. Therefore it is not known to EIDC2.

$incoming_Z(n) = \{$edges $e \in Z | Target(e) = n\}$.

This denotes all edges in the graph Z with n as their target. We say that $incoming_E(n) \approx incoming_F(n)$ if the node n has the same corresponding incoming edges with the same weights in both graphs, i.e. $e' = (a,n) \in incoming_F(n) \Leftrightarrow e = (a,n) \in incoming_E(n)$ and $\forall e \in incoming_E(n) : weight(e) = weight(e')$.

We say that a node n is *complete* iff the following holds:

1) The node n has already been processed by EIDC2.
2) In the iteration where n was chosen and processed by EIDC2, when PROCESSPOSREQ() is about to be called, $incoming_E(n) \approx incoming_F(n)$.

The second condition states that all incoming edges to n that are supposed to be derived by EIDC in this entire increment were either (a) present when n was chosen for processing or (b) added as a result of PROCESSCOND() or PROCESSNEGREQ() in the same iteration. Consequently these edges could be completely processed through PROCESSPOSREQ() in that iteration. Thus, EIDC2 will find all incoming edges to n that can be derived by FastIDC derivations.

A node X is *ready* iff every node $n \in after(X)$ is *complete*.

Suppose some node X is *unprocessed* and *ready*. Then $\forall e \in Incoming(C,X) : Source(e) \notin unprocessed$, so X is a candidate for processing by EIDC2.

Lemma 1: Consider an iteration of EIDC2 where a node X that is *ready* is chosen and popped from *unprocessed*. Then X will be *complete* after this iteration.

A consequence is that after X is processed, all nodes ordered before it will be one step closer to becoming ready.

Proof: We will examine incoming edges to X in F and show that by the time EIDC2 calls PROCESSPOSREQ(), each such edge must have a corresponding edge in E with identical weight. We will prove this by considering all such edges in F from the perspective of their type and the derivation rule used to derive them.

Conditional edges in F derived by D1: Suppose edge $e' = (A,X)$ in F was derived by D1, where X is the node we are currently processing. In Figure 2 this means that X corresponds to C in rule D1. We want to show that a corresponding edge $e = (A,X)$ in E is derived by EIDC2.

Since the rule was applicable in F, there was a contingent edge of weight $-x$ from a node B to X (and consequently $B \in \mathit{after}(X)$). A contingent edge is always added externally and never by derivations. It must have been present in F before the increment where rule D1 was applied, and since we are considering an equivalent series of calls to FastIDC and EIDC2, it must also have been present in E before the corresponding increment started. A similar argument applies to the edge of weight y from X to B.

There was also a positive requirement edge from A to B in F with weight v, or the rule would not have been applicable. We know X is *ready* and $B \in \mathit{after}(X)$, and therefore B is *complete*. By the definition of *complete*, the corresponding edge from A to B existed before PROCESSPOSREQ() was executed for B. Then all preconditions for applying rule D1 in E were satisfied in that increment, and the edge $e = (A,X)$ was derived by PROCESSPOSREQ() at that time.

The same edge still exists before the call to PROCESSPOSREQ() when processing X.

Conditional edges in F derived by D2–D3 and D5: A conditional edge in F cannot have been added "externally" (between calls to FastIDC) but must have been added by a derivation rule. Rule D1, which we have already considered, can create a conditional edge to a node X where no such edge previously existed. Apart from this rule, only rules D2–D3 and D5 can create conditional edges to a node X, and in this case there must already exist a conditional edge to node X before the rule is applied. Therefore any chain of derivations of conditional edges must start with an application of rule D1.

The derived edges in such a chain always point towards the same target node (the source of the contingent constraint) whereas the source of the derived edges moves "backwards" along a positive edge. For example, suppose we apply rule D3 in Figure 2 with $A = X$. This requires an existing conditional edge (C,X) and generates a new conditional edge (D,X), where both edges have the same target X but the source changes from C to D "backwards" along a positive edge of weight v. In D3 and D5 the new weight is directly affected by the weight of that positive edge, while in D2 the weight of the negative edge is used.

Figure 6 shows an example of such a chain of derivations. The topmost graph is the starting state from which the bold edges in the lower graph, labeled with derivation rules, are added by derivations: First $D1 :< A, -55 >$, then $D3 :< A, -45 >$, then $D2 :< A, -39 >$, then $D3 :< A, -29 >$. We can see that the helper function PROCESSCOND() derives all edges in such a derivation chain, making sure only to derive along shortest paths since the derived edges along these paths are the tightest possible.

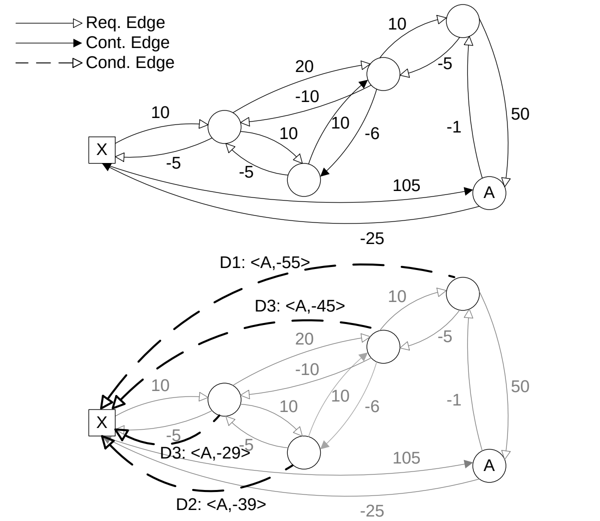

Fig. 6. Example derivation chain.

Consider what happens when X is processed by EIDC2. Since each of the involved positive edges (and any involved contingent edge) targets a node that is in $\mathit{after}(X)$, and since X is *ready*, they are all present by the time X become current and PROCESSCOND() is called. Therefore all the edges corresponding to derivations in F by derivation rules D2–D3 and D5 will be derived by EIDC2 before the call to PROCESSPOSREQ() as is required by the lemma.

Requirement edges in F derived by D4 and D6–D9: Requirement edges can be derived by derivation rules D4, D6–D9. Rules D8 and D9 cannot be the cause of deriving edges that are part of F but not E since we already saw that any conditional edge must be in both graphs and D8-D9 are applied directly to these by both algorithms. If we look at the remaining rules, D4 and D6-D7, these exactly match the rules D2-D3 and D5 edge-for-edge with the only difference being the type of edge, conditional vs. requirement. Because of this these rules are handled by EIDC2 inside PROCESSNEGREQ() similarly to how the previous rules are handled inside PROCESSCOND().

So we see that all edges targeting X in F must have corresponding edges targeting X in E. Hence, $\mathit{incoming}_E(X) \approx \mathit{incoming}_F(X)$. We also recognize that this equality is satisfied while processing X before PROCESSPOSREQ() since this helper function was not required by the proof of the lemma. Therefore all requirements in the lemma are satisfied. ■

We now continue with the next part of the completeness proof. In the following we will view derivation rules D8-D9 as part of the application which generated the conditional edge they operate on. This allow us to present a lemma on a general property of derivation:

Lemma 2: A new incoming edge to a node C can only be created if its EDG contains a positive edge (A,B) and a negative edge (B,C). The created edge (by derivations in F and helper functions in E) then becomes (A,C).

Proof: This is obvious upon inspection of the derivation rules in Figure 2. Notice that the type of the edges does not matter, only the sign of their weights. ∎

For the next lemma we first define the *depth* of a node as the number of edges in the longest path to it which consists of only negative edges. The kind of edge (conditional, requirement or contingent) does not matter. The depth of a node is well-defined since there are no negative cycles present (detection is handled by the CCGraph). We will have reason to use depth both in E and F and will use the graph designation as subscript to separate these depths.

Lemma 3: When a node, X is chosen for processing by EIDC2, it has received its final depth in E, $depth_E(X)$.

Proof: Suppose a node X is chosen by EIDC2 when it has $depth_E(X) = k$. The only way $depth_E(X)$ can change during or after processing is if X receives a new incoming negative edge. By lemma 2 this requires a node A with a negative edge (A, X). Due to how nodes are chosen in the algorithm, A must already have been processed before X was chosen. Then the new incoming edge would already have been created through PROCESSPOSREQ() and so cannot affect the depth after X was chosen for processing. ∎

It is possible to infer from the reasoning in the proof that the execution order between nodes (as entailed by negative edges) is never found by calls to PROCESSCOND() *nor* PROCESSNEGREQ() *but only by the* PROCESSPOSREQ() *helper function. The former calls however are important since they make sure that derivations only follow shortest paths. Therefore, both phases of derivation are needed.*

We now define the set containing all nodes that have negative edges targeting X.

$IncNegSrc_Z(X) = \{\text{nodes } n | \exists e = (n, X) \in Z \text{ which is negative}\}$.

Lemma 4: For every node X that is selected for processing by EIDC2 the following holds:

1) X is *ready* (every node $n \in after(X)$ is *complete*)
2) $depth_E(X) = depth_F(X)$
3) $IncNegSrc_E(X) = IncNegSrc_F(X)$

Proof: Induction over the depth of the nodes in the graph.

Basis: Any node X that has $depth_E(X) = 0$ and is chosen for processing is *ready*. This follows since $depth_E(X) = 0$ means that there is no incoming negative edge to X. Thus there were no incoming edge at the start of the increment. The same is then true for X in F (assuming that E and F were identical at the start of the increment). By lemma 2 this means that there cannot be an incoming negative edge to X in F. By the definition we then have $after(X) = \{\text{nodes } n \mid \text{there is a path of negative edges from } n \text{ to } X \text{ in } F\} = \emptyset$. Hence every node in $after(X)$ is complete, making X *ready*.

Furthermore we have $depth_F(X) = depth_E(X) = 0$ (since X had no incoming negative edges in any of the graphs). This means that we also have $IncNegSrc_E(X) = IncNegSrc_F(X)$. Thus the corresponding nodes have the same negative incoming edges, so the theorem holds for every node X with depth 0.

Induction assumption: The theorem holds for all nodes with $depth_E < k$.

Induction step: Suppose a node X with $depth_E(X) = k$ is chosen for processing. All $n \in IncNegSrc_E(X)$ must then have $depth_E(n) < k$, since otherwise $depth_E(X) > k$. Given how the algorithm chooses nodes, we know that all $n \in IncNegSrc_E(X)$ have already been processed and (by lemma 3) have received their final depths. By the induction hypothesis they were *ready* when processed and hence for each node n, $p \in after(n) \Rightarrow p$ is *complete*. By Lemma 1 the nodes themselves are now *complete*. Therefore any ordering derived through the nodes in $IncNegSrc_E(X)$ has been done by PROCESSPOSREQ(). Therefore we have $IncNegSrc_E(X) = IncNegSrc_F(X)$. Then all nodes in $after(X)$ are *complete* and hence X is *ready*.

By induction the theorem holds for nodes with any depth and hence for every node selected for processing. ∎

Theorem 1: The EIDC2 algorithm is correct.

Proof: First, EIDC2 is sound. It is also complete since by lemma 4 all nodes chosen for processing are *ready* and by lemma 1 become *complete* after processing. This means that all nodes in E have the same incoming edges as the corresponding nodes in F. This also guarantees that all nodes that need to be processed become *active* (and are processed). Therefore all corresponding edges match between the graphs. Since EIDC2 also performs all the tests (exactly the ones done by FastIDC, which is correct) needed to correctly verify the DC state of the STNU, we conclude that it is correct. ∎

C. Complexity

As the previous section shows nodes are chosen in the optimal order, i.e. there is no need to process a node more than once. Therefore, there will be $O(n)$ outer iterations. We can divide the work done in inner iterations into two parts. One which deals with conditional edges and one which deals with requirement edges. The part which deals with requirement edges is shown [7] to have a complexity in $O(n^2)$ in each iteration. The part which deals with conditional edges is shown to have an accumulated complexity in $O(n^3)$ over all outer iterations. So in total, the execution of the EIDC2 algorithm for a modified STNU will have a complexity in $O(n^3)$.

VIII. WORST-CASE EXAMPLE REVISITED

TABLE II. EDGES DERIVED IN THE EXAMPLE.

Step	Proc.	*unprocessed*	*active*	Added edges
1	-	{a-f,X-Z}	{a}	$b \to a$
2	a	{b-f,X-Z}	{a-c,e}	$b \to c$, $b \to e$
3	b	{c-f,X-Z}	{a-e}	$d \to c$, $d \to e$, $f \to c$, $f \to e$
4	c	{d-f,X-Z}	{a-e,X}	$f \to X$, $d \to X$
5	d	{e-f,X-Z}	{a-e,X}	$Y \to e$
6	e	{f,X-Z}	{a-e,X,Z}	$f \to Z$, $Y \to Z$
7	Z	{f,X-Y}	{a-e,X-Z}	$f \to X$ (second edge)
8	Y	{f,X}	{a-e,X-Z}	$\ldots \to X$
9	X	{f}	{a-e,X-Z}	...

To see how much difference the strategy used by EIDC2 makes we will revisit the worst-case example of EIDC. In the figures we will use nodes with dashed background to capture that they are in *active*, and as before bold font to show they are in *unprocessed* and gray color for the processed nodes. Table II shows the way the algorithm handles the scenario. The scenario starts as in Figure 3, but in step 5 EIDC2 is

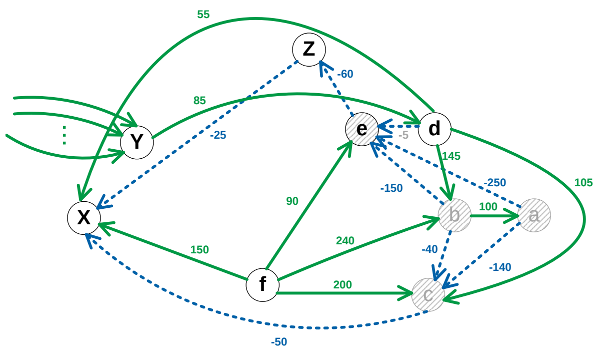

Fig. 7. Scenario where X was chosen by EIDC.

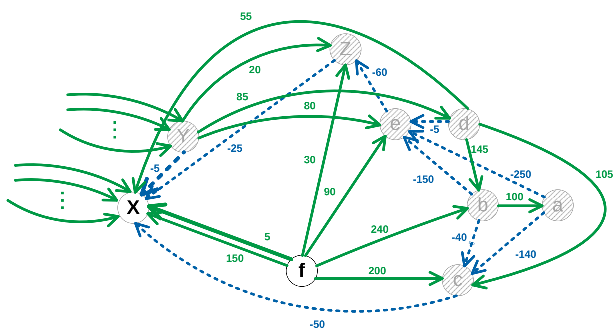

Fig. 8. Final situation in the example, where X is chosen by EIDC2.

faced with the situation in Figure 7. Here X is in *unprocessed*, but it cannot be chosen for processing since Z is after X and also in *unprocessed*. By waiting until Z is processed, X is protected against processing before any node whose relation to X is determined by interaction with Z, in this case Y. Figure 8 shows the final situation when X is chosen for processing.

IX. Related and Future Work

Recently several papers [12], [13] have examined the use of Timed Game Automata (TGA) for both verification and execution of STNUs. These solutions work on a smaller scale and do not exploit the inherent structure of STNUs as distance graphs. Therefore they are more useful in networks that are small in size but involve choice and resources which cannot be handled by pure STNU algorithms.

Future work includes investigating how to execute the resulting EDG, perhaps through use of recent work [14].

During the review of this paper Morris [15] presented a new $O(n^3)$ algorithm for full DC verification. The algorithm has the following properties: **(1)** It uses FastIDC derivations (called "plus-minus"). **(2)** It calculates distances in reverse direction (cf. DDG graph). **(3)** It uses a distance-limited version of Dijkstra over positive edges. **(4)** Derivation continues only after all incoming negative edges are processed (through recursion). **(5)** Non-DC is detected as cycles consisting of only negative edges (cf. CCGraph). Although Morris uses a different notation [5] and labeled graphs which allow derivations in any order (EIDC2 does Cond/NegReq/PosReq), we are convinced that the algorithms in fact are equivalent. Had we not focused on the incremental usage this might have been the conclusion of this paper. From the work of Morris we conclude that EIDC2 handles full DC verification by starting from an unprocessed EDG where all nodes are added to *active*. It remains future work to rigorously compare the algorithms and investigate their relative performance in practice.

X. Conclusion

We have presented EIDC2, an improvement over the previously fastest algorithm (EIDC) for incremental DC verification of STNUs. Whereas EIDC had a worst-case of $\Omega(n^4)$ for one increment (shown in this paper), EIDC2 lowers this to $O(n^3)$. This is possible since EIDC2 uses a different ordering technique that allows the processing of incoming positive requirement edges to a node to be used for determining the optimal processing order between nodes. EIDC2 also makes use of the processing techniques put forward by EIDC to avoid repeatedly discovering tighter and tighter edges.

A consequence of the $O(n^3)$ DC verification is that any system that today relies on consistency checks for STNs may at no increase in worst case run-time extend their representation to model uncontrollable durations.

Acknowledgments

This work is partially supported by the Swedish Research Council (VR) Linnaeus Center CADICS, the ELLIIT network organization for Information and Communication Technology, the Swedish Foundation for Strategic Research (CUAS Project), the EU FP7 project SHERPA (grant agreement 600958), and Vinnova NFFP6 Project 2013-01206.

References

[1] R. Dechter, I. Meiri, and J. Pearl, "Temporal Constraint Networks," *Artificial Intelligence*, vol. 49, no. 1-3, pp. 61–95, 1991.

[2] T. Vidal and M. Ghallab, "Dealing with Uncertain Durations in Temporal Constraint Networks Dedicated to Planning," in *Proc. ECAI*, 1996.

[3] P. Morris, N. Muscettola, and T. Vidal, "Dynamic Control of Plans with Temporal Uncertainty," in *Proc. IJCAI*, 2001.

[4] P. Morris and N. Muscettola, "Temporal dynamic controllability revisited," in *Proc. AAAI*, 2005.

[5] P. Morris, "A Structural Characterization of Temporal Dynamic Controllability," in *Proc. CP*, 2006.

[6] J. L. Stedl, "Managing Temporal Uncertainty Under Limited Communication: A Formal Model of Tight and Loose Team Coordination," Master's thesis, Massachusetts Institute of Technology, 2004.

[7] M. Nilsson, J. Kvarnström, and P. Doherty, "EfficientIDC: A Faster Incremental Dynamic Controllability Algorithm," in *Proc. ICAPS*, 2014.

[8] M. Nilsson, J. Kvarnström, and P. Doherty, "Classical Dynamic Controllability Revisited: A Tighter Bound on the Classical Algorithm," in *Proc. ICAART*, 2014.

[9] L. Hunsberger, "Fixing the Semantics for Dynamic Controllability and Providing a more Practical Characterization of Dynamic Execution Strategies," in *Proc. TIME*, 2009.

[10] J. Stedl and B. Williams, "A Fast Incremental Dynamic Controllability Algorithm," in *Proc. ICAPS Workshop on Plan Execution*, 2005.

[11] M. Nilsson, J. Kvarnström, and P. Doherty, "Incremental Dynamic Controllability Revisited," in *Proc. ICAPS*, 2013.

[12] A. Cimatti, L. Hunsberger, A. Micheli, and M. Roveri, "Using Timed Game Automata to Synthesize Execution Strategies for Simple Temporal Networks with Uncertainty," in *Proc. AAAI*, 2014.

[13] A. Cesta, A. Finzi, S. Fratini, A. Orlandini, and E. Tronci, "Analyzing Flexible Timeline-based Plans," in *Proc. ECAI*, 2010.

[14] L. Hunsberger, "A Faster Execution Algorithm for Dynamically Controllable STNUs," in *Proc. TIME*. IEEE, 2013, pp. 26–33.

[15] P. Morris, "Dynamic Controllability and Dispatchability Relationships," in *Proc. CPAIOR*, 2014.