A New Approach to Checking the Dynamic Consistency of Conditional Simple Temporal Networks

Luke Hunsberger[1] and Roberto Posenato[2]

[1] Vassar College, Poughkeepsie, NY 12604, USA,
hunsberger@vassar.edu,
[2] University of Verona, Verona, Italy,
roberto.posenato@univr.it,

Abstract. A Conditional Simple Temporal Network (CSTN) is a structure for representing and reasoning about temporal constraints in domains where constraints may apply only in certain scenarios. Observations in real time incrementally reveal the "real" scenario. A CSTN is *dynamically consistent* (DC) if there is a strategy for executing its time-points that guarantees the satisfaction of all relevant constraints. The fastest DC-checking algorithm for CSTNs is based on constraint propagation. This paper introduces a new approach to DC checking for CSTNs, inspired by controller-synthesis algorithms for Timed Game Automata. The new algorithm views the DC-checking problem as a two-player game, searching an abstract game tree to find a "winning" strategy, using Monte-Carlo Tree Search and Limited Discrepancy Search to guide its search. An empirical evaluation shows that the new algorithm is competitive with the propagation-based algorithm.

1 Introduction

Recently, there have been significant advances in the theory and practice of temporal networks and Timed Game Automata (TGAs). Of particular interest is the work by Cimatti et al. [5, 4] which showed that dynamic consistency/controllability (a.k.a., DC-checking) problems for a variety of temporal networks can be reduced to controller-synthesis problems for TGAs. Although their work revealed strong theoretical connections between temporal networks and TGAs, it did not immediately produce practical algorithms because the generic TGA solver (UPPAAL-TIGA [3, 2]) could not exploit the particular structure of the DC-checking problem. However, more recently, Cimatti et al. [6] presented a practical DC-checking algorithm for one kind of network—a Disjunctive Temporal Network with Uncertainty (DTNU)—by using the temporal-network-to-TGA reduction from the earlier work, but customizing the controller-synthesis algorithm to exploit the structure of the DC-checking problem for DTNUs.

Inspired by their approach, this paper presents a new DC-checking algorithm for a different kind of network: a Conditional Simple Temporal Network (CSTN). Although the spirit of the approach is similar, especially in the traversal of the abstract *simulation graph,* the substantial differences between the execution semantics for CSTNs and DTNUs required the development of numerous novel representations and algorithmic techniques. Furthermore, the new algorithm does not reduce CSTNs to TGAs; instead, it maps TGA-based techniques into the realm of temporal networks, using

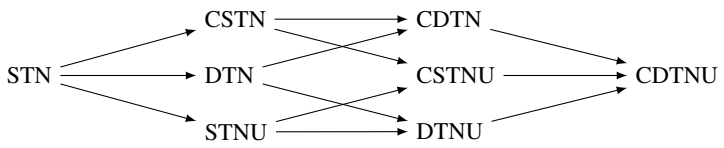

Fig. 1: A hierarchy of temporal networks from least to most expressive

basic consistency-checking algorithms from the literature on Simple Temporal Networks (STNs) [8], whereas Cimatti et al. use techniques from Satisfiability Modulo Theory (SMT) [1]. In addition, the new algorithm uses Monte-Carlo Tree Search (MCTS) [9] and Limited Discrepancy Search (LDS) [10] to guide its search. An empirical evaluation of the new algorithm demonstrates that it is competitive with the propagation-based algorithm due to Hunsberger et al. [13] that is currently the fastest known DC-checking algorithm for CSTNs. Although the propagation-based algorithm tends to be faster for weakly constrained networks and over-constrained (or inconsistent) networks, the performance of the two algorithms is quite similar for moderately constrained networks. In addition, the new algorithm is able to efficiently process particular worst-case structures that dramatically slow down the propgation-based algorithm.

2 Background

A *Simple Temporal Network* (STN) is a structure for representing and reasoning about time [8]. An STN includes real-valued variables called time-points (frequently notated as $X, Y, Z, \ldots$), and binary difference constraints on those time-points (e.g., $Y - X \leq 5$). The graph for an STN is a pair $(\mathcal{T}, \mathcal{E})$, where each constraint, $Y - X \leq \delta$ in $\mathcal{C}$, corresponds to an edge in $\mathcal{E}$ from X to Y with length δ. An STN is *consistent* if there exists an assignment of values to its time-points that together satisfy all of its constraints. The consistency-checking problem for STNs can be solved in cubic time by computing the *distance matrix* $\mathcal{D}$ for the STN graph (i.e., its *all-pairs shortest-paths* matrix: for each $X, Y \in \mathcal{T}$, $\mathcal{D}(X, Y)$ equals the length of the shortest path from X to Y).

Theorem 1 (Decomposability of STNs [8, 11]). *Let $\mathcal{S} = (\mathcal{T}, \mathcal{C})$ be any STN, and $\mathcal{D}$ its distance matrix. For any $\mathcal{X} \subseteq \mathcal{T}$, let $\mathcal{C}|_{\mathcal{X}} = \{(Y - X \leq \mathcal{D}(X, Y)) \mid X, Y \in \mathcal{X}\}$ be the* all-pairs, shortest-paths *constraints from $\mathcal{C}$ that involve time-points in $\mathcal{X}$. Then any solution for the STN $\mathcal{S}|_{\mathcal{X}} = (\mathcal{X}, \mathcal{C}|_{\mathcal{X}})$ can be extended to form a solution for $\mathcal{S}$; and the distance matrix $\mathcal{D}|_{\mathcal{X}}$ for $\mathcal{S}|_{\mathcal{X}}$ satisfies: for all $X, Y \in \mathcal{X}$, $\mathcal{D}|_{\mathcal{X}}(X, Y) = \mathcal{D}(X, Y)$.*

Henceforth, $\mathcal{D}|_{\mathcal{X}}$ will be called the *restriction* of $\mathcal{D}$ to time-points in $\mathcal{X}$.

More Expressive Temporal Networks. The expressiveness of an STN can be extended in several independent dimensions, such as: (C) allowing *conditional* constraints; (D) allowing *disjunctive* constraints; and (U) allowing temporal intervals with *uncertain* durations. Providing different combinations of these features results in temporal networks with names such as *Conditional Simple Temporal Networks* (CSTNs), *Disjunctive Temporal Networks with Uncertainty* (DTNUs), and so on. (For networks that allow disjunctive constraints, the "simple" modifier is dropped.) Arranging these networks according to their expressiveness leads to the hierarchy in Fig. 1.

Dynamic Consistency/Controllability. Although the presence of disjunctive constraints makes the consistency-checking problem for Disjunctive Temporal Networks (DTNs) NP-hard [8], the execution semantics for DTNs is the same as that for STNs. In contrast, the presence of conditional constraints (C) or intervals with uncertain durations (U) dramatically changes the execution semantics and, therefore, requires new notions of consistency (or controllability). For example, a CSTN has conditional constraints that may apply only in certain scenarios; and the particular scenario that obtains is only incrementally revealed through the execution of *observation time-points.* A CSTN is *dynamically consistent* if there exists a *dynamic strategy* for executing its time-points such that all relevant constraints will be satisfied no matter which scenario is incrementally revealed [18]. On the other hand, a Simple Temporal Network with Uncertainty (STNU) includes intervals with uncertain durations, represented by *contingent links.* The ending time-points of contingent links are *uncontrollable,* but guaranteed to fall within certain bounds. An STNU is *dynamically controllable* if there exists a dynamic strategy for executing its controllable time-points such that all constraints will be satisfied no matter how the uncertain durations turn out [17]. Finally, the dynamic controllability of CSTNUs and CDTNUs, which allow conditional constraints *and* contingent links, has also been defined [12, 4]. (When including both conditional constraints and contingent links, the term "controllability" is preferred.) Crucially, the decisions made by *dynamic* execution strategies must only depend on past information, whether gleaned from the execution of observation time-points or the observed durations of contingent links.

DC-checking algorithms. An algorithm for checking the dynamic consistency or controllability of a temporal network is called a *DC-checking* algorithm. Morris [16] recently presented a cubic-time DC-checking algorithm for STNUs. However, the consistency-checking problem for DTNs and the DC-checking problem for all of the other network classes in Fig. 1 are known to be NP-hard.[3]

Cimatti et al. [4] showed that the DC-checking problem for CDTNUs, the most expressive network in Fig. 1, can be reduced to a controller-synthesis problem for TGAs, but the resulting algorithm was not practical because the generic controller-synthesis algorithm for TGAs could not exploit the DC-checking problem structure. More recently, Cimatti et al. [6] presented a practical DC-checking algorithm for DTNUs, using the same temporal-network-to-TGA reduction, but implementing a customized controller-synthesis algorithm that exploits the structure of the DC-checking problem for DTNUs.

Inspired by their approach, this paper presents a new DC-checking algorithm for CSTNs. Although similar in spirit, the new algorithm differs substantially from the DTNU algorithm. For example: (1) instead of translating the input CSTN into a TGA, the new algorithm performs all computations on related STNs; (2) although it too traverses a *simulation graph* involving subsets of already-scheduled time-points, the transitions in that graph are completely different owing to the completely different execution semantics for CSTNs; (3) it uses unions of STNs instead of logic-based formulas to represent the so-called *winning regions;* and (4) it uses Monte-Carlo Tree Search (MCTS) [9] and Limited Discrepancy Search (LDS) [10] to guide its search.

[3] Dechter et al. [8] for DTNs; Comin and Rizzi [7] for CSTNs; the rest follow from these results.

3 Conditional Simple Temporal Networks

This section reviews the dynamic consistency of CSTNs [18], using definitions drawn from Hunsberger et al. [13]. To begin, let $\mathcal{P}$ be a set of propositional letters. A *label* is any consistent conjunction of (positive or negative) literals from $\mathcal{P}$; the set of all such labels is denoted by $\mathcal{P}^*$; and the empty label is denoted by $\boxdot \in \mathcal{P}^*$.

A CSTN may include time-points and temporal constraints that apply only in certain scenarios. The "real" scenario is incrementally revealed through the execution of *observation time-points* (ObsTPs). (For convenience, *non-observation* time-points may be called *ordinary* time-points (OrdTPs).) Each observation time-point $P?$ has a corresponding propositional letter p; executing $P?$ generates a truth value for p. During execution, the information that is incrementally gleaned from such observations is recorded in a label called the *current partial scenario* (CPS). For example, if p and s have been observed to be $true$, and q has been observed to be $false$, then the CPS would be the label $p\neg qs$.

Time-points and constraints in a CSTN may have propositional labels. For example, the labeled constraint $(Y - X \leq 5, p\neg q)$ specifies that $Y - X \leq 5$ must hold in any (partial or complete) scenario that is consistent with $p\neg q$.

Definition 1 (CSTN) *A* Conditional Simple Temporal Network *(CSTN) is a tuple,* $\langle \mathcal{T}, \mathcal{C}, L, \mathcal{OT}, \mathcal{O}, \mathcal{P} \rangle$*, where:*

- $\mathcal{P}$ *is a finite set of propositional letters;*
- $\mathcal{T}$ *is a finite set of real-valued variables (time-points);*
- $\mathcal{C}$ *is a finite set of labeled constraints, each having the form,* $(Y - X \leq \delta, \ell)$*, where* $X, Y \in \mathcal{T}$*,* $\delta \in \mathbb{R}$*, and* $\ell \in \mathcal{P}^*$*;*
- $L : \mathcal{T} \rightarrow \mathcal{P}^*$ *is a function assigning labels to time-points;*
- $\mathcal{OT} \subseteq \mathcal{T}$ *is a set of observation time-points; and*
- $\mathcal{O} : \mathcal{P} \rightarrow \mathcal{OT}$ *is a bijection between ObsTPs and propositional letters.*

A CSTN typically includes a special time-point Z whose value is fixed at 0; all other time-points are constrained to occur at or after Z. (If no such Z exists, one may be added without adverse effects.) Binary constraints involving Z are equivalent to unary constraints. For example, $Z - X \leq -5$ is equivalent to $X \geq 5$.

Each CSTN $\mathcal{S} = \langle \mathcal{T}, \mathcal{C}, L, \ldots \rangle$, has an associated graph, $\langle \mathcal{T}, \mathcal{E} \rangle$, where the edges in $\mathcal{E}$ correspond to the labeled constraints in $\mathcal{C}$. In particular, each $(Y - X \leq \delta, \ell) \in \mathcal{C}$ corresponds to an edge from X to Y annotated by the *labeled value* $\langle \delta, \ell \rangle$. A sample CSTN graph is shown in Fig. 2. This graph includes structures, called *negative q-loops,* that can dramatically slow down the DC-checking algorithm of Hunsberger et al. [13].

The execution semantics for a CSTN can be expressed in terms of a two-player game between the agent responsible for executing its time-points and the environment responsible for selecting truth values for propositional letters [4]. An execution *run* begins with $Z = 0$ and an empty current partial scenario. At any time, the agent may choose to execute any time-point whose label is entailed by the CPS. Whenever an observation time-point $P?$ is executed, the environment must instantaneously select a truth value for the corresponding letter p. If $p = true$, then p is conjoined to the CPS; otherwise, $\neg p$ is conjoined to the CPS. The execution run is completed whenever it happens that all time-points whose labels are entailed by the CPS have been executed. If

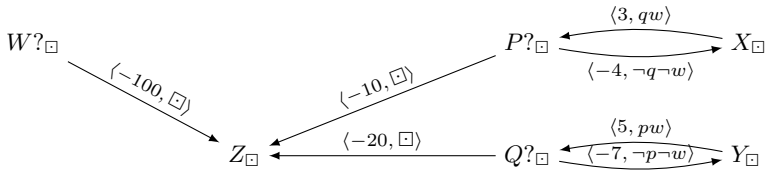

Fig. 2: A CSTN with negative q-loops

all constraints whose labels are consistent with the final CPS are satisfied, then the agent wins; otherwise, the environment wins. A sample winning run for the CSTN from Fig. 2 is given below, where the choices made by the environment are parenthesized.

$$Z = 0; W? = 100\ (\neg w); X = 101; Y = 106; P? = 107\ (\neg p); Q? = 125\ (q).$$

Since the final CPS is $\neg pq\neg w$, constraints labeled by pw, qw and $\neg q\neg w$ do not apply.

Definition 2 (Scenario) *A* scenario *over a set $\mathcal{P}$ of propositional letters is a function, $s : \mathcal{P} \to \{true, false\}$. For each label $\ell \in \mathcal{P}^*$, the truth value for ℓ determined by s is denoted by $s(\ell)$. The set of all scenarios over $\mathcal{P}$ is denoted by $\mathcal{I}$. A* partial scenario *is any function, $s : \mathcal{P}' \to \{true, false\}$, where $\mathcal{P}' \subseteq \mathcal{P}$.*

Definition 3 (Schedule) *A* schedule *for a set of time-points $\mathcal{T}$ is a (complete) mapping, $\psi : \mathcal{T} \to \mathbb{R}$. The set of all schedules over all* subsets *of $\mathcal{T}$ is denoted by Ψ.*

Definition 4 (Projection) *Let $\mathcal{S} = \langle \mathcal{T}, \mathcal{C}, L, \ldots \rangle$ be any CSTN, and s any scenario over $\mathcal{P}$. The* projection *of $\mathcal{S}$ onto s—notated $\mathcal{S}(s)$—is the STN, $(\mathcal{T}_s^+, \mathcal{C}_s^+)$, where:*

- $\mathcal{T}_s^+ = \{T \in \mathcal{T} \mid s(L(T)) = true\}$*; and*
- $\mathcal{C}_s^+ = \{(Y - X \leq \delta) \mid \text{for some } \ell, (Y - X \leq \delta, \ell) \in \mathcal{C} \text{ and } s(\ell) = true\}$

Definition 5 (Execution Strategy) *An* execution strategy *for a CSTN $\mathcal{S}$ is a mapping $\sigma : \mathcal{I} \to \Psi$, such that for each scenario $s \in \mathcal{I}$, the domain of $\sigma(s)$ is $\mathcal{T}_s^+$. If, in addition, for each scenario s, the schedule $\sigma(s)$ is a solution to the projection $\mathcal{S}(s)$, then σ is called* viable. *The execution time for any X in $\sigma(s)$ is denoted by $[\sigma(s)]_X$.*

Definition 6 (History) *Let $\mathcal{S} = \langle \mathcal{T}, \mathcal{C}, L, \ldots \rangle$ be any CSTN, s any scenario, σ any execution strategy for $\mathcal{S}$, and t any real number. The* history *of t in the scenario s, for the strategy σ—notated $Hist(t, s, \sigma)$—is the set of observations made before time t according to $\sigma(s)$:* $Hist(t, s, \sigma) = \{(p, s(p)) \mid P? \in \mathcal{T}_s^+ \text{ and } [\sigma(s)]_{P?} < t\}$.

Definition 7 (Dynamic Execution Strategy) *An execution strategy σ for a CSTN is called* dynamic *if for any scenarios s_1 and s_2, and any time-point X:*

let: $t = [\sigma(s_1)]_X$; *if:* $Hist(t, s_1, \sigma) = Hist(t, s_2, \sigma)$; *then:* $[\sigma(s_2)]_X = t$.

Definition 8 (Dynamic Consistency) *A CSTN $\mathcal{S}$ is* dynamically consistent *(DC) if there exists an execution strategy for $\mathcal{S}$ that is both dynamic and viable.*

Several DC-checking algorithms for CSTNs have been presented in the literature [18, 4, 13, 7]. However, only the propagation-based algorithm due to Hunsberger et al. [13] has been empirically demonstrated to be practical. It uses six rules for propagating labeled constraints in the CSTN graph. Although its worst-case complexity is conjectured to be exponential, it stands as the fastest DC-checking algorithm for CSTNs so far.

4 Overview of Our Approach

Following Cimatti et al. [4], our approach views the execution of time-points in a CSTN as a two-player game between the agent who executes time-points and the environment that assigns truth values to propositions. Each *run* of the game consists of a sequence of turns. Since the environment is idle until an observation time-point is executed, it is useful to model a turn for the agent as involving the (typically not simultaneous) execution of zero or more ordinary time-points, followed by the execution of a single observation time-point $P?$. A turn for the environment involves instantaneously assigning a truth value to p, the propositional letter associated with $P?$, resulting in either p or $\neg p$ being appended to the current partial scenario. The run ends when all time-points whose labels are entailed by the CPS have been executed. The agent wins if all constraints entailed by the final CPS are satisfied. The agent seeks a winning strategy (i.e., a strategy that guarantees that all relevant constraints will be satisfied no matter which truth values the environment chooses along the way). The agent's strategy can be *dynamic* in that it can react to observations in real time, but only after some positive delay.

Since there are $n!$ possible orders in which to execute n time-points, and each time-point may be assigned any number of values, searching for a winning strategy by exploring the associated game tree is not practical. To make the search space finite, we use a more abstract representation for the agent's moves, one that does *not* specify execution times for the time-points being "played". Each (abstract) move is represented by a pair $(\chi, P?)$, where χ is a possibly empty set of ordinary time-points, and $P?$ is an observation time-point. To retain maximal flexibility, the time-points in $\chi \cup P?$ are only partially ordered. In particular, for each $X \in \chi$, and each as-yet-unplayed $Y \notin \chi \cup P?$, the ordering constraints, $X \leq P? \leq Y$, are accumulated. Thus, a typical sequence of alternating moves has the following form:

$$(\chi_1, P_1?), (p_1 = b_1), (\chi_2, P_2?), (p_2 = b_2), \ldots, (\chi_k, P_k?), (p_k = b_k), \chi_{k+1}$$

where the χ_i are disjoint sets of ordinary time-points, the $P_i?$ are distinct observation time-points, the p_i are the corresponding propositional letters, and the b_i are truth values. The associated ordering constraints can be concisely expressed as follows:

$$Z = 0 \leq \chi_1 \leq P_1? \leq \chi_2 \leq P_2? \leq \ldots \leq \chi_k \leq P_k? \leq \chi_{k+1}$$

where expressions of the form $\chi_i \leq P_i? \leq \chi_{i+1}$ stand for the *sets* of constraints, $(\forall X \in \chi_i)(X \leq P_i?)$ and $(\forall X \in \chi_{i+1})(P_i? \leq X)$. No ordering constraints are imposed among any pair of time-points, X and Y, that belong to the same set χ_i.

Consider the CSTN graph shown in Fig. 3. One possible sequence of (abstract) moves is: $(\{X\}, P?), (p = \mathit{true}), \{Y\}$. The corresponding ordering constraints are: $Z \leq X \leq P? \leq Y$. Another possible sequence is: $(\emptyset, P?), (p = \mathit{false}), \{X, W\}$. The corresponding ordering constraints are: $Z \leq P? \leq \{X, W\}$.

The (abstract) game tree—also called the *simulation graph* [3]—is a branching tree with finitely many nodes. It includes: (1) *agent nodes* (Agt-nodes) that represent abstract states where it is the agent's turn to "play" time-points; and (2) *environment nodes* (Env-nodes) that represent abstract states where it is the environment's turn to select truth values for propositional letters. Each move for the agent is represented by an edge

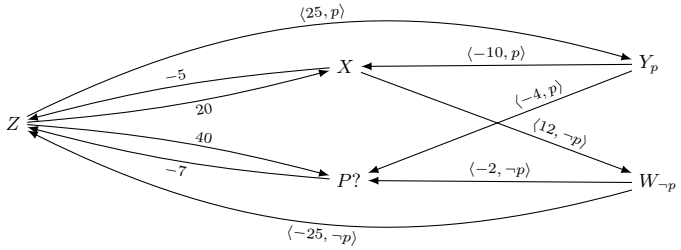

Fig. 3: A sample CSTN

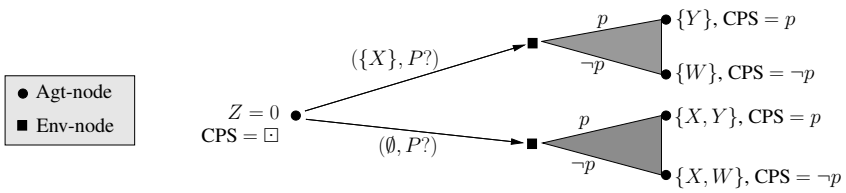

Fig. 4: The abstract game tree (i.e., simulation graph) for the CSTN from Fig. 3

from an Agt-node to an Env-node. Each *binary choice* available to the environment is represented by a *hyper-edge* [15] whose source is an Env-node, and whose target set is a *pair* of Agt-nodes: one for *true,* one for *false.* The game tree for the CSTN from Fig. 3 is illustrated in Fig. 4, where the shaded triangles represent hyper-edges.

When it is the agent's turn, the number of available moves depends on the number of as-yet-unplayed time-points whose labels are entailed by the current partial scenario. When it is the environment's turn, there are always exactly two available moves: $true$ or $false$. For each move $(\chi, P?)$ by the agent, our algorithm effectively asks the question: "Is there an assignment for the time-points in $\chi \cup P?$ that can force a win *from this point onward?*" That question is not answered immediately. But notice that a win-forcing assignment for time-points in $\chi \cup P?$ must be able to force a win whether the environment subsequently chooses $true$ or $false$ for p. Thus, when it is the agent's turn, the agent need only find one win-forcing move out of all of its available moves, but that move must be win-forcing for *both* branches arising from the environment's two choices. And, of course, this property continues recursively as the game tree is descended.

Although the question, "Is there a win-forcing assignment?", is not answered immediately, the constraints from the CSTN that apply to the time-points in $\chi \cup \{P?\}$ (and any other already-played time-points) together with the ordering constraints discussed above, do restrict the space within which such win-forcing assignments must reside—should they exist. This restricted space of possible win-forcing assignments can be represented by an STN, hereinafter called the *current STN.* A consistency check on the current STN can be used to prune moves that cannot be part of a winning strategy.

Our algorithm has two interleaved phases: a forward phase that uses a modified depth-first search to find a sequence of moves terminating in a leaf node whose current STN is consistent; and a backward phase to be described later. Recall that the game ends when all observation time-points whose labels are entailed by the CPS have been

	Z	X	$P?$
Z	0	20	40
X	-5	0	35
$P?$	-7	0	0

(a) $\mathcal{D}_1$

	Z	X	$P?$	Y
Z	0	15	21	25
X	-5	0	16	20
$P?$	-7	0	0	18
Y	-15	-10	-4	0

(b) $\mathcal{D}_p$

	Z	X	$P?$	W
Z	0	20	30	32
X	-13	0	27	12
$P?$	-13	0	0	12
W	-25	-2	-2	0

(c) $\mathcal{D}_{\neg p}$

	Z	X	$P?$
Z	0	15	21
X	-13	0	16
$P?$	-13	0	0

(d) $\mathcal{D}_1^*$

Fig. 5: Distance matrices related to the running example

executed. At that point, the player's last move is to schedule all remaining ordinary time-points whose labels are entailed by the CPS. Let χ be the set of such time-points. Note that for a terminal node whose current STN is consistent, the question "Is there an assignment for the time-points in χ that can force a win *from this point onward*?" is trivially "Yes". In particular, any solution to that node's current STN will work. In other words, for a terminal node in the game tree, the solution set for the current STN represents not only the restricted domain within which any win-forcing assignment must reside, but in fact the actual set of win-forcing assignments for that node.

The root node of the game tree from Fig. 4 has the following information:

- Z is the only time-point that has been "played";
- $\pi_0 = \boxdot$ is the current partial scenario;
- $\mathcal{T}_0 = \{Z, X, P?\}$ are the time-points whose labels are entailed by $\pi_0 = \boxdot$; and
- $\mathcal{C}_0 = \mathcal{C}|_{\{Z,X,P?\}} = \{X \in [5, 20], P? \in [7, 40]\}$, the CSTN constraints whose labels are entailed by $\pi_0 = \boxdot$. (The constraints in $\mathcal{C}_0$ have had their labels removed.)

From the root node, there are only two legal moves: $(\emptyset, P?)$ or $(\{X\}, P?)$. Let's explore the latter move. After this move, the current STN is $\mathcal{S}_1 = (\{Z, X, P?\}, \mathcal{C}_0 \cup \theta)$, where:

- $\theta = \{Z \leq X \leq P?\}$, the ordering constraints associated with the move $(\{X\}, P?)$.

The distance matrix $\mathcal{D}_1$ for $\mathcal{S}_1$ is shown in Fig. 5a. Any win-forcing assignment to time-points in $\{Z, X, P?\}$ must satisfy the constraints represented by this matrix; however, at this point, it is not known whether such an assignment exists.

Next, suppose that the environment chooses to set $p = true$. In that case, we get:

- $\{Z, X, P?\}$ are the time-points that have been played;
- $\pi_1^p = p$ is the current partial scenario; and
- $\mathcal{T}_1^p = \{Y\}$ is the set of unplayed time-points whose labels are entailed by $\pi_1^p = p$.

Since there are no observation time-points in $\mathcal{T}_1^p$, the agent has only one option: to play all of the time-points in $\mathcal{T}_1^p$. The resulting current STN is $\mathcal{S}_1^p = (\mathcal{T}_1^p, \mathcal{C}_1^p \cup \theta_p)$, where:

- $\mathcal{C}_1^p = \mathcal{C}|_{\{Z,X,P?,Y\}} = \mathcal{C}_0 \cup \{Y \leq 25, X - Y \leq -10, P? - Y \leq -4\}$; and
- $\theta_p = \theta \cup \{P? \leq Y\}$, the relevant ordering constraints.

The distance matrix, $\mathcal{D}_p$, for this STN is shown in Fig. 5b. Because this distance matrix is consistent, it represents the space of win-forcing assignments for that terminal node. However, recall that win-forcing means forcing a win *from this point onward.* We have

to back-propagate to see whether preceding nodes can get to this point. By Theorem 1, any solution to the *restriction* of $\mathcal{D}_p$ to the time-points in $\{Z, X, P?\}$ can be extended to a solution to $\mathcal{D}_p$ over the time-points in $\{Z, X, P?, Y\}$. Thus, any solution to the upper-lefthand 3×3 sub-matrix of $\mathcal{D}_p$ is a win-forcing assignment for $\{Z, X, P?\}$. That is, if Z, X and $P?$ are assigned values that satisfy that sub-matrix, and the environment chooses $p = \mathit{true}$, then we can choose a value for Y that will yield a winning state.

However, that is not enough. We must also ensure that a winning state is arrived at should the environment choose $p = \mathit{false}$. The relevant matrix for that case is $\mathcal{D}_{\neg p}$, shown in Fig. 5c. Its upper-lefthand 3×3 matrix must be satisfied by Z, X and $P?$ to ensure a win should the environment choose $p = \mathit{false}$. Since we want to ensure a win regardless of which choice the environment makes, a win-forcing assignment for Z, X and $P?$ must satisfy *both* matrices. Thus, the relevant matrix is the *element-wise minimum* of the corresponding upper-lefthand 3×3 sub-matrices of $\mathcal{D}_p$ and $\mathcal{D}_{\neg p}$. That matrix, $\mathcal{D}_1^*$, is shown in Fig. 5d. The matrix $\mathcal{D}_1^*$ is the *back-propagation* of $\mathcal{D}_p$ and $\mathcal{D}_{\neg p}$.

Now, in general, at any given node, the agent typically has many available moves. Some of these moves might yield sets of win-forcing assignments; and each such set can be represented by an STN, as described above. However, the *union* of such win-forcing sets cannot usually be represented by a single STN; instead, a union of STNs—or their corresponding distance matrices—is needed. Thus, the back propagation described above must be able to accommodate unions of distance matrices at each node. Fortunately, in practice, back-propagation is done incrementally; so, only the most recently generated winning distance matrix needs to be processed. For example, suppose that the p branch of a hyper-edge emanating from some Env-node yields a new winning matrix, $\mathcal{D}^p$, while the $\neg p$ branch already has a union of winning matrices, $\mathcal{D}_1^{\neg p} \cup \ldots \cup \mathcal{D}_k^{\neg p}$. Then the new win-set for that Env-node is given by the union, $(\mathcal{D}^p \wedge \mathcal{D}_1^{\neg p}) \cup \ldots \cup (\mathcal{D}^p \wedge \mathcal{D}_k^{\neg p})$, where $\wedge$ denotes the "element-wise minimum" operator.

Overall, the algorithm propagates forward through the game tree searching for a terminal node whose current STN is consistent. The solution space for that STN is the set of win-forcing assignments for that node. The algorithm then *back-propagates* that win-set through the game tree to determine whether it may yield non-empty win-sets for any predecessor nodes. If the algorithm ever finds a non-empty win-set for the root node, it reports that the network is dynamically consistent. Since the algorithm only seeks the existence of a non-empty win-set for the root node, not the maximum win-set, it frequently creates and explores only a small fraction of the tree.

For any given node, the win-sets must be subsets of the solution space for the current STN. That is, the current STN represents necessary, but typically not sufficient constraints on win-forcing assignments. The following properties of win-sets provide the rationale for the algorithm's handling of winning distance matrices.

(P1) Let v be an Agt-node; and $v_1, \ldots, v_k$ its child Env-nodes. If $\mathcal{W}_1, \ldots, \mathcal{W}_k$ are win-sets for those child nodes, and $\mathcal{T}_v$ is the set of time-points whose labels are entailed by the CPS at node v, then $(\mathcal{W}_1|_{\mathcal{T}_v}) \cup \ldots \cup (\mathcal{W}_k|_{\mathcal{T}_v})$ is a win-set for v.

(P2) Let v be an Env-node; and v_p and $v_{\neg p}$ its child Agt-nodes. If $\mathcal{W}_p$ and $\mathcal{W}_{\neg p}$ are win-sets for v_p and $v_{\neg p}$, respectively, and $\mathcal{T}_v$ is the set of time-points whose labels are entailed by the CPS at node v, then $(\mathcal{W}_p|_{\mathcal{T}_v}) \cap (\mathcal{W}_{\neg p}|_{\mathcal{T}_v})$ is a win-set for v.

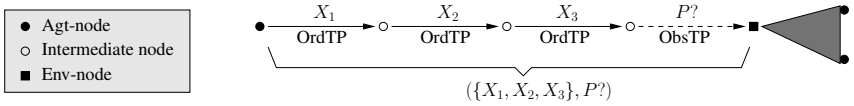

Fig. 6: A sequence of edges representing a transition from an Agt-node to an Env-node

(P1) follows from Theorem 1. (P2) follows from the independence of the environment's choices: a win-forcing assignment for v must be win-forcing for *both* v_p and $v_{\neg p}$.

Incrementally constructing agent moves. From any given Agt-node, there may be a large number of agent moves, each of the form $(\chi, P?)$, where χ is a possibly empty subset of OrdTPs and $P?$ is an ObsTP. For example, if there are k OrdTPs and m ObsTPs whose labels are entailed by the current partial scenario, then there are $m2^k$ possible moves of the form $(\chi, P?)$. Thus, it is impractical to compute all of these potential moves, let alone explore them. For this reason, the new DC-checking algorithm *incrementally* constructs such moves in a sequence of *steps.* In each step, a time-point is selected for incorporation into the nascent move. Each OrdTP that is selected is added to χ. When an ObsTP is (eventually) selected, the construction of the move $(\chi, P?)$ is completed.

Thus, a move $(\chi, P?)$ from an Agt-node to an Env-node is represented by a sequence of *step-edges,* as illustrated in Fig. 6. If an OrdTP is being played, the step-edge is a *forward OrdTP edge* and the destination is an *intermediate node.* If an ObsTP is being played, the step-edge is a *forward ObsTP edge* and the destination is an Env-node. In this way, each agent move is represented by a sequence of zero or more forward OrdTP edges, followed by a single ObsTP edge that terminates at an Env-node.

Combining Monte-Carlo Tree Search and Limited Discrepancy Search. For each Agt-node, the algorithm maintains a queue of as-yet-unexplored step-edges. At each step, the algorithm uses Monte-Carlo Tree Search [9] to determine the "best" time-point to incorporate into the move $(\chi, P?)$ being incrementally constructed. When MCTS selects an ObsTP, the move is completed and the resulting Env-node is created and explored.

To avoid getting trapped in an unpromising portion of the search space, instead of doing ordinary depth-first search, the algorithm uses *Limited Discrepancy Search* (LDS) [10], as follows. First, each call to MCTS generates an ordered list of step-moves. If the best step-move is explored, there is no penalty (or discrepancy). But if the algorithm backtracks and tries the next best move, a penalty of $+1$ is accumulated. Further backtracking leading to exploring even worse moves leads to higher penalties. Thus, the sequence of moves that is currently being explored, going all the way back to the root node, has an associated total path discrepancy. LDS with a limit of L ignores any move that would lead to a total path discrepancy greater than L. Since a winning strategy might not be obtained from LDS using a given limit L, the algorithm employs an *iterative deepening* version of discrepancy search where the limit L starts out at 0 (i.e., only best step-moves are explored at each step). If that search fails to find a winning strategy, the limit L is incremented and the algorithm tries again. This process continues until a winning strategy is found or the search space is exhausted.

Pseudo-code for the new DC-checking algorithm. The new DC-checking algorithm for CSTNs is called `SG-DC-CHECK` (for "simulation-graph DC-checking"). Pseudo-code for the `SG-DC-CHECK` algorithm is given in Tables 1 and 2. The `SG-DC-CHECK`

```
SG-DC-CHECK:
  v_0 := root Agt-node;
  v_0.queue := MCTS(v_0), a sorted list of step-moves, each of the form, stepMove(v_0, TP);
  L := 0    (initial discrepancy limit);
  while(true)
    result := SEARCH(v_0, 0, L).
    if (result == non-empty win-set for v_0), then return DC;
    elseif (result == search_space_exhausted), then return non-DC;
    else L := L + 1.

SEARCH(v, d, L):    (v = Agt-node, d = accumulated discrepancy)
  new_winners := NIL
  while((new_winners == NIL) && (v.queue ≠ ∅))
    stepMove(v_c, TP) := pop(v.queue);    (v_c is an Agt or intermed. node, TP is a time-point)
    when(d + discr(stepMove(v_c, TP)) ≤ L)
      new_winners := PROCESS-EDGE(v, stepMove(v_c, TP), d).
  return new_winners.
```

Table 1: Pseudo-code for the SG-DC-CHECK algorithm for CSTNs (Part One)

algorithm calls the SEARCH method on the root Agt-node v_0, whose sorted list of initial step-moves has been generated by Monte-Carlo Tree Search and pushed onto v_0's queue. The SEARCH method processes (forward or backward) edges on the queue of as-yet-unprocessed moves. The behavior of the PROCESS-EDGE method depends on the type of edge (forward or backward; OrdTP, ObsTP or Env) to be processed. Processing a forward OrdTP edge creates a new intermediate node; and uses Monte-Carlo Tree Search to generate a sorted list of legal step-moves to be pushed onto the parent Agt-node's queue. Processing a forward ObsTP edge creates a new Env-node and pushes an (Env) hyper-edge onto the parent Agt-node's queue. The first time an (Env) hyper-edge is processed, two Agt-node children are created. If both child nodes happen to be leaf nodes (i.e., terminal nodes), then their restricted win-sets are intersected and a bkwdObsEdge (i.e., backward observation edge) is pushed onto the parent Agt-node's queue to initiate back-propagation of that new restricted win-set; otherwise, the (Env) hyper-edge that is currently being processed is pushed back onto the queue to ensure that it is immediately re-visited. Whenever an (Env) hyper-edge is re-visited, it calls the SEARCH method on one of the child Agt-nodes to see whether a new win-set can be generated. If so, then a bkwdEnvEdge (i.e., backward environment edge) is pushed onto the queue to back-propagate that new win-set. If the back-propagation peters out (i.e., fails to reach the root node of the entire simulation graph), then the (Env) hyper-edge will be re-visited to see whether additional win-sets for the child Agt-nodes might be computed.

As already indicated, back-propagation is implemented by processing bkwdEnvEdges and bkwdObsEdges. Processing a bkwdEnvEdge for a given Env-node v_E takes a new win-set for one of the child Agt-nodes $\hat{v}$ and intersects it with the existing win-sets for the other child Agt-node $\tilde{v}$. If any new win-sets are generated that are not subsumed by any of the existing win-sets for v_E, then those win-sets are packaged into a bkwdObsEdge that is pushed onto the queue to ensure that back-propagation continues. Processing a bkwdObsEdge takes new win-sets for an Env-node v_E and restricts the corresponding

distance matrices to the time-points relevant to the parent Agt-node. If new win-sets for the parent Agt-node are generated, then a bkwdEnvEdge is pushed onto the queue of the Env-node parent of that Agt-node, and back-propagation continues. If the back-propagation ever reaches the root Agt-node with a non-empty win-set, then the network is declared to be DC. If the search fails to find a win-set for the root node, then the discrepancy limit L is increased. That process continues until a win-set for the root node is found, or the search space is exhausted.

5 Empirical Evaluation

The fastest DC-checking algorithm for CSTNs from the literature is the propagation-based algorithm of Hunsberger et al. [13] which, for convenience, shall be called the DC-CHECK algorithm. This section compares the performance of the new SG-DC-CHECK algorithm and the existing DC-CHECK algorithm.

First, it must be acknowledged that the DC-CHECK algorithm performs very well on networks that are either (1) very loosely constrained or (2) inconsistent. In the first case, there is not much constraint propagation to perform, so the algorithm runs quickly. In the second case, negative loops signalling non-DC can often be found very quickly. In contrast, the SG-DC-CHECK algorithm typically explores an exponential number of nodes in the simulation graph even for loosely constrained networks, because it must ensure the existence of a win-forcing strategy; and for non-DC networks, it must *exhaust* the search space to prove that no winning strategy can exist. Therefore, this section focuses primarily on moderately constrained networks.

The generation of CSTN instances was based on random workflow schema generated by the ATAPIS toolset [14]. It is hoped that instances generated in this way may be closer to examples that might be encountered in the real world. Thirty consistent CSTNs were randomly generated. Each CSTN had between 107 and 155 time-points, between 8 and 16 observation time-points, and between 428 and 970 constraints. Both algorithms, DC-CHECK and SG-DC-CHECK, were implemented in Lisp and run on Intel Core i7-4790 machines with 3.6 GHz processors (4 cores/8 threads), running Ubuntu 14.04.04 LTS (kernel version 3.16.0-67-generic) and Allegro Common Lisp (ACL) Enterprise Edition, version 8.1. For increased efficiency (i.e., to dramatically reduce the number of redundant labeled edges stored during constraint propagation), the DC-CHECK algorithm stored labeled edges in subsumption hierarchies, one hierarchy for each pair of time-points. Because the DC-CHECK algorithm is deterministic, running it repeatedly on the same network yields essentially identical timing results. Therefore, it was run only once on each network. In contrast, the SG-DC-CHECK algorithm employs randomness, both from the use of Monte-Carlo Tree Search to determine "best" moves, and in deciding which of the two child nodes to explore when processing an (Env) hyper-edge. Therefore, its timing results can vary when run repeatedly on the same network. Thus, SG-DC-CHECK was run five times on each network.

The average run-time of the SG-DC-CHECK algorithm was (statistically significantly) better than that of the DC-CHECK algorithm for 11 of the 30 instances, while the DC-CHECK algorithm was faster on 18 of the 30 instances. For one instance, *both* algorithms timed out (over 10 minutes). Of the 18 instances where the DC-CHECK

algorithm was faster, the SG-DC-CHECK algorithm timed out on 6 instances. Over the 23 instances where neither algorithm timed out, the mean run times were 40.0 seconds for DC-CHECK and 53.8 seconds for SG-DC-CHECK, with standard deviations of 80.8 and 89.6, respectively. Furthermore, over the 29 instances where the DC-CHECK algorithm did not time out, the mean run time of DC-CHECK was 38.4 seconds, with a standard deviation of 80.7, but over the same 29 instances, the *minimum* run time of the two algorithms (actual for DC-CHECK, average for SG-DC-CHECK over 5 trials) had a mean of 22.1 seconds with a standard deviation of 24.0. This suggests that running the two algorithms in parallel, and taking the answer obtained by whichever algorithm finishes sooner, has the potential to dramatically improve the timing performance.

Finally, to demonstrate a shortcoming of the DC-CHECK algorithm, recall the small CSTN from Fig. 2. The loops between X and $P?$, and Y and $Q?$, are examples of what are called *negative q-loops*. They do *not* necessarily cause a network to be non-DC. In fact, the CSTN in Fig. 2 *is* DC. However, they can lead to an incredible amount of constraint propagation. Each loop reinforces the other, gradually increasing the lower bounds on $P?$ and $Q?$ until they eventually approach the value of the lower bound on $W?$, which is 100. (To understand why, see the constraint-propagation rules presented by Hunsberger et al. [13].) If the lower bound on $W?$ is increased to 1000, the DC-CHECK algorithm takes almost three minutes to determine that the network is DC. In contrast, the SG-DC-CHECK algorithm solves this network in 80 milliseconds, regardless of the size of the lower bound on $W?$. This example highlights the fact that the two algorithms have very different strengths and weaknesses. It suggests that further study of both algorithms is warranted. Both algorithms may achieve superior results in the future by improving their implementations. For example, improving the consistency checking and tuning the parameters governing the Monte-Carlo Tree Search may dramatically improve the performance of the SG-DC-CHECK algorithm.

6 Related Work

The SG-DC-CHECK algorithm continues a line of research that dates back to the literature on finding fixed-points for dependency graphs [15], synthesizing controllers for TGAs [3], and checking the dynamic controllability of DTNUs [6]. Like the DC-checking algorithm for DTNUs, the SG-DC-CHECK algorithm: (1) searches through an abstract *simulation graph;* (2) manages its traversal through that graph by keeping track of edges waiting to be processed, computing winning sets for nodes, and keeping track of dependencies among nodes; (3) keeps track of ordering constraints and performs consistency checks to prune nodes during search; and (4) in successful instances, can be used to generate a suitable controller (i.e., a dynamic execution strategy). However, adapting the high-level approach to CSTNs required the development of significant novel representations and techniques. Some differences include: (1) TGAs are not used at all; (2) winning sets are represented by STNs, not TGA clock zones; (3) consistency checking is done using STN algorithms, not Satisfiability Modulo Theory; (4) the simulation graph includes hyper-edges that represent the *true* and *false* branches for observations; (5) STN conjunction and restriction are used to back-propagate winning sets; (6) Monte-Carlo Tree Search and Limited Discrepancy Search are used to guide the search.

References

1. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, pp. 825–885. IOS Press (2009)
2. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K., Lime, D.: UPPAAL-TIGA: Time for playing games! In: Damm, W., Hermanns, H. (eds.) 19th International Con- ference on Computer Aided Verification (CAV07), pp. 121–125. No. 4590 in Lecture Notes in Computer Science, Springer-Verlag, Berlin Heidelberg (2007)
3. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) Proceedings of CONCUR 2005–Concurrency Theory. pp. 66–80. IEEE Computer Society Press, United States (2005)
4. Cimatti, A., Hunsberger, L., Micheli, A., Posenato, R., Roveri, M.: Sound and complete algorithms for checking the dynamic controllability of temporal networks with uncertainty, disjunction and observation. In: TIME-2014. pp. 27–36. IEEE Computer Society (Sept 2014)
5. Cimatti, A., Hunsberger, L., Micheli, A., Roveri, M.: Using timed game automata to synthesize execution strategies for simple temporal networks with uncertainty. In: Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI-2014) (2014)
6. Cimatti, A., Micheli, A., Roveri, M.: Dynamic controllability of disjunctive temporal networks: Validation and synthesis of executable strategies. In: Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI-2016) (2016)
7. Comin, C., Rizzi, R.: Dynamic consistency of conditional simple temporal networks via mean payoff games: a singly-exponential time DC-checking. In: 22st International Symposium on Temporal Representation and Reasoning (TIME 2015). pp. 19–28. IEEE CPS (Sep 2015)
8. Dechter, R., Meiri, I., Pearl, J.: Temporal constraint networks. Artificial Intelligence 49(1-3), 61–95 (1991), http://dx.doi.org/10.1016/0004-3702(91)90006-6
9. Gelly, S., Silver, D.: Monte-carlo tree search and rapid action value estimation in computer go. Artificial Intelligence 175, 1856–1875 (July 2011)
10. Harvey, W., Ginsberg, M.: Limited discrepancy search. In: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence. pp. 607–613 (1995)
11. Hunsberger, L.: Group Decision Making and Temporal Reasoning. Ph.D. thesis, Harvard University (2002), available as Harvard Technical Report TR-05-02
12. Hunsberger, L., Posenato, R., Combi, C.: The Dynamic Controllability of Conditional STNs with Uncertainty. In: Proceedings of the Workshop on Planning and Plan Execution for Real-World Systems: Principles and Practices (PlanEx) at ICAPS-2012. pp. 1–8 (2012)
13. Hunsberger, L., Posenato, R., Combi, C.: A sound-and-complete propagation-based algorithm for checking the dynamic consistency of conditional simple temporal networks. In: 22st International Symposium on Temporal Representation and Reasoning (TIME 2015). pp. 4–18. IEEE CPS (Sep 2015)
14. Lanz, A., Reichert, M.: Enabling time-aware process support with the atapis toolset. In: Limonad, L., Weber, B. (eds.) Proceedings of the BPM Demo Sessions 2014. CEUR Workshop Proceedings, vol. 1295, pp. 41–45. CEUR (2014)
15. Liu, X., Smolka, S.A.: Simple linear-time algorithms for minimal fixed points (extended abstract). In: Proceedings of the 25th International Colloquium on Automata, Languages and Programming. pp. 53–66. ICALP '98, Springer-Verlag, London, UK, UK (1998)
16. Morris, P.: Dynamic controllability and dispatchability relationships. In: Integration of AI and OR Techniques in Constraint Programming, LNCS, vol. 8451, pp. 464–479. Springer (2014)
17. Morris, P.H., Muscettola, N., Vidal, T.: Dynamic control of plans with temporal uncertainty. In: Nebel, B. (ed.) Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01). pp. 494–502. Kaufmann (2001)
18. Tsamardinos, I., Vidal, T., Pollack, M.E.: CTP: A new constraint-based formalism for conditional, temporal planning. Constraints 8, 365–388 (2003)

PROCESS-EDGE(v, (OrdTP)$stepMove(v_c, X), d$): (X is an OrdTP)
Create intermediate node with X played; and accumulate (partial) ordering constraints;
when(constraints consistent)
push(MCTS(v_c), $v.queue$) (i.e., push sorted list of legal step-moves onto v's queue).

PROCESS-EDGE(v (ObsTP)$stepMove(v_c, P?), d$):
Create Env-node v_E, and compute distance matrix $\mathcal{D}_E$ for its current STN;
when($\mathcal{D}_E$ is consistent),
push((Env) hyper-edge($v_E, P?$), $v.queue$).

PROCESS-EDGE(v, (Env) hyper-edge($v_E, P?$), d):
if(first time visiting this hyper-edge):
Create child Agt-nodes, v_p and $v_{\neg p}$, and corresponding distance matrices, $\mathcal{D}_p$ and $\mathcal{D}_{\neg p}$;
when(element-wise min. $\mathcal{D}'_E$ of restricted matrices $\mathcal{D}'_p$ and $\mathcal{D}'_{\neg p}$ is consistent):
for each $\hat{v} \in \{v_p, v_{\neg p}\}$,
if($\hat{v}$ is a leaf node), $\hat{v}.finished := true$, and $\hat{v}.winners := \mathcal{D}'_E$;
else, $\hat{v}.queue :=$ MCTS($\hat{v}$);
if ($v_p.finished == v_{\neg p}.finished == true$), push(bkwdObsEdge($v_E, \mathcal{D}'_E$), $v.queue$);
else push((Env) hyper-edge($v_E, P?$), $v.queue$);
else (i.e., re-visiting this edge):
if($v_p.finished == v_{\neg p}.finished == true$), then return NIL;
else select $\hat{v} \in \{v_p, v_{\neg p}\}$ such that $\hat{v}.finished =$ NIL;
when $winner :=$ SEARCH($\hat{v}$) $\neq \emptyset$,
push(bkwdEnvEdge($v_E, \hat{v}, winner$), $v.parent.queue$).

PROCESS-EDGE(v,bkwdEnvEdge($v_E, \hat{v}, newWinner$), d): (d ignored during back-prop)
$\tilde{v} :=$ the *other* Agt-node child of v_E (i.e., $\tilde{v}$ and $\hat{v}$ are the children of v_E);
$winAcc := \emptyset$; ($winAcc$ will accumulate winners for v_E)
for each $oldWinner \in \tilde{v}.winners$,
$possWinner := newWinner \cap oldWinner$;
if $possWinner \not\subseteq \bigcup_{W \in v_E.winners} W$, push($possWinner, accWinners$);
push ((Env) hyper-edge($v_E, P?$), v); (if back-prop sputters, must revisit hyper-edge)
if($winAcc \neq \emptyset$), push(bkwdObsEdge($v_E, winAcc$), $v.queue$); (continue back-prop)

PROCESS-EDGE(v,bkwdObsEdge($v_E, newWinners$), d): (d ignored during back-prop)
$winAcc := \emptyset$; ($winAcc$ will accumulate winners for v)
for each$newWinner \in newWinners$,
$restrictedWinner := newWinner|_{v.\mathcal{T}}$ (i.e., restrict winner to time-points in v)
if $restrictedWinner \not\subseteq \bigcup_{W \in v.winners} W$, push($restrictedWinner, winAcc$);
if((($winAcc \neq \emptyset$) && ($rootNode?(v)$))) return DC; (Non-empty win-set for root node!)
else if($winAcc \neq \emptyset$) push(bkwdEnvEdge($v_E, winAcc$), $v.queue$). (continue back-prop)

Table 2: Pseudo-code for the SG-DC-CHECK algorithm for CSTNs (Part Two)