Faster Dynamic-Consistency Checking for Conditional Simple Temporal Networks

BLIND REVIEW

Abstract

A Conditional Simple Temporal Network (CSTN) is a structure for representing and reasoning about time in domains where temporal constraints may be conditioned on outcomes of observations made in real time. A CSTN is dynamically consistent (DC) if there is a strategy for executing its time-points such that all relevant constraints will necessarily be satisfied no matter which outcomes happen to be observed. The literature on CSTNs contains only one sound-and-complete DC-checking algorithm that has been implemented and empirically evaluated. It is a graph-based algorithm that propagates labeled constraints/edges. A second algorithm has been proposed, but not evaluated. It aims to speed up DC checking by more efficiently dealing with so-called *negative q-loops.*

This paper presents a new two-phase approach to DC-checking for CSTNs. The first phase focuses on identifying negative q-loops, labeling key time-points within such loops. The second phase focuses on computing (labeled) distances from each time-point to a single sink node. The new algorithm, which is also sound and complete for DC-checking, is then empirically evaluated against both pre-existing algorithms and shown to be much faster across not only previously published benchmark problems, but also a new set of benchmark problems. The results show that, on DC instances, the new algorithm tends to be an order of magnitude faster than both existing algorithms. On all other benchmark cases, the new algorithm performs better than or equivalently to the existing algorithms.

Introduction

A Conditional Simple Temporal Network (CSTN) is a data structure for reasoning about time in domains where some constraints may apply only in certain scenarios. For example, a patient who tests positive for a certain disease may need to receive care more urgently than someone who tests negative. Conditions in a CSTN are represented by propositional letters whose truth values are not controlled, but instead *observed* in real time. Just as doing a blood test generates a positive or negative result that is only learned in real time, the execution of an *observation time-point* in a CSTN generates a truth value for its corresponding propositional letter. An execution strategy for a CSTN specifies when the time-points will be executed. A strategy can be *dynamic* in that its decisions can react to information from past observations. A CSTN is said to be dynamically consistent (DC) if it admits a dynamic strategy that guarantees the satisfaction of all relevant constraints no matter which outcomes are observed during execution.

Different varieties of the DC property have been defined that differ in how reactive a strategy can be. Tsamardinos *et al.* (2003) stipulated that a strategy can react to an observation after any arbitrarily small, positive delay. Comin *et al.* (2015) defined ϵ-DC, which assumes that a strategy's reaction times are bounded below by a fixed $\epsilon > 0$. Finally, Cairo et al. (2016) defined π-DC, which allows a dynamic strategy to react instantaneously (i.e., after zero delay).

This paper focuses exclusively on the π-DC property. Cairo et al. (2016) presented the first sound-and-complete π-DC-checking algorithm. However, their algorithm, which is (pseudo) singly-exponential in the number of propositional letters, has never been implemented or evaluated empirically. Hunsberger and Posenato (2018) presented an alternative algorithm, which we call $\pi\mathtt{DC}_{18}$, that is based on the propagation of labeled constraints. An empirical evaluation of their algorithm demonstrated its practicality. Noting that the $\pi\mathtt{DC}_{18}$ algorithm can get bogged down, repeatedly cycling through certain graphical structures, called *negative q-loops,* Hunsberger and Posenato (2019) presented (what we call) the $\pi\mathtt{DC}_{19}^{\infty}$ algorithm, which included a rule that can generate edges labeled by expressions such as $\langle -\infty, \alpha\rangle$. They generalized existing rules to accommodate such edges. Although they did not empirically evaluate the $\pi\mathtt{DC}_{19}^{\infty}$ algorithm, it is expected to deal more efficiently with negative q-loops.

This paper presents a new approach to DC-checking for CSTNs that involves two phases. The first phase focuses on identifying negative q-loops and labeling key *time-points*—not edges—within such loops with expressions such as $\langle -\infty, \alpha\rangle$. The second phase focuses on computing (labeled) distances from each time-point to a single sink node. The new algorithm, which is also sound and complete for DC-checking, is then empirically evaluated against both pre-existing algorithms and shown to be much faster across not only previously published benchmark problems, but also a new set of benchmark problems. The results show that, on DC instances, the new algorithm tends to be an order of magnitude faster than both existing algorithms. On all other benchmark cases, the new algorithm performs better than or equivalently to the existing algorithms.

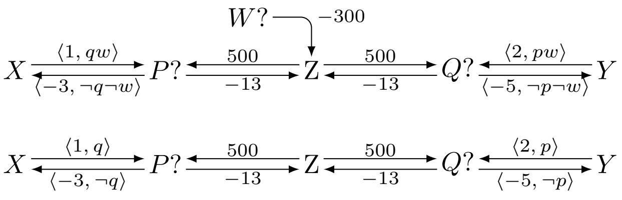


Figure 1: Two sample CSTN graphs

Preliminaries

Dechter *et al.* (1991) introduced Simple Temporal Networks (STNs) to facilitate reasoning about time. An STN has real-valued variables, called *time-points,* and binary difference constraints on those variables. Most STNs have a time-point Z whose value is fixed at zero. A *consistent* STN is one that has a solution as a constraint satisfaction problem.

Tsamardinos *et al.* (2003) presented CSTNs, which augment STNs to include *observation time-points* and their associated *propositional letters.* In a CSTN, the execution of an observation time-point $P?$ generates a truth value for its associated letter p. In addition, each time-point can be labeled by a conjunction of literals specifying the scenarios in which that time-point must be executed. Finally, they noted that for any reasonable CSTN, the propositional labels on its time-points must satisfy certain properties.

Hunsberger et al. (2012) generalized CSTNs to include labels on constraints, and formalized the properties held by *well-defined* CSTNs. Then Cairo *et al.* (2017) showed that for *well-defined* CSTNs, no loss of generality results from *removing* the labels from its time-points. Therefore, this paper restricts attention to CSTNs whose time-points have no labels—the so-called *streamlined* CSTNs—and henceforth uses the term CSTN to refer to streamlined CSTNs.

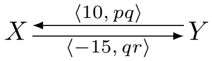
Fig. 1 shows two sample CSTNs in their graphical forms, where nodes represent time-points, and labeled, directed edges represent conditional binary difference constraints. For example, in the top figure, $\mathrm{Z} = 0$; and $P?, Q?$ and $W?$ are observation time-points whose execution generates truth values for p, q and w, respectively. The edge from Y to $Q?$ being labeled by $\langle 2, pw \rangle$ indicates that the constraint, $Q? - Y \leq 2$ applies only in scenarios where p and w are both $\top$.

The Dynamic Consistency of CSTNs

Since the execution of an observation time-point $P?$ generates a truth value for its associated letter p, a *dynamic execution strategy* can *react* to observations, in real time, possibly making different execution decisions in different scenarios. A *dynamically consistent* CSTN is one that has an execution strategy that guarantees that all relevant constraints will be satisfied no matter which values are observed in real time. This paper focuses on π*-dynamic* strategies, which can react *instantaneously* to observations (Cairo, Comin, and Rizzi 2016). The full set of definitions is given in the Appendix.

Existing π-DC-Checking Algorithms

This paper restricts attention to the π-DC-checking problem for CSTNs (i.e., execution strategies can react instantaneously to observations). For convenience, we use the term DC to mean π-DC. The π-DC-checking algorithms discussed in this paper are all based on the propagation of labeled constraints. In graphical terms, each algorithm employs a set of rules for generating new edges from existing edges in the CSTN graph. Whereas the characteristic feature of an inconsistent STN is the existence of a negative-length loop, the characteristic feature of a non-DC CSTN is the existence of a negative-length loop whose edges have mutually consistent propositional labels. For example, a CSTN with the loop shown below



must be non-DC since in any scenario consistent with pqr, both constraints along the negative loop must be satisfied, which is impossible. (Such a loop might only be revealed after extensive constraint propagation.) However, a DC CSTN may contain negative-length loops whose edges have mutually *inconsistent* propositional labels; they are called *negative q-loops.* For example, the CSTN at the top of Fig. 1 is DC, despite having two negative q-loops: one from X to $P?$ to X, and one from Y to $Q?$ to Y. The key difference is that in these loops, the propositional labels are inconsistent (e.g., the label pw on the edge from Y to $Q?$ is inconsistent with the label $\neg p \neg w$ on the edge from $Q?$ to Y), so they don't *necessarily* imply that the network is non-DC. However, negative q-loops are not always benign. Therefore, negative q-loops pose important challenges for any π-DC-checking algorithm.

Each algorithm in this paper generates new edges in the CSTN graph until: (1) a negative-length self-loop (i.e., a negative-length edge from a node to itself) with a consistent label is generated, or (2) no new edges can be generated. In case (1), the network is not DC; in case (2), it is DC.

The $\pi\mathtt{DC}_{18}$ Algorithm

The only sound-and-complete π-DC-checking algorithm that has been implemented and empirically evaluated in the literature is the π-DC-Check algorithm of Hunsberger and Posenato (2018), hereinafter called $\pi\mathtt{DC}_{18}$. To deal with constraints having inconsistent labels, the algorithm sometimes generates a new kind of propositional label, called a *q-label.*

Definition 1 (Q-literals, q-labels). A *q-literal* has the form $?p$, where $p \in \mathcal{P}$. A q-literal represents that a proposition's value is *unknown.* A *q-label* is a conjunction of literals and/or q-literals. $\mathcal{Q}^*$ denotes the set of all q-labels.

For example, $p(?q)\neg r$ and $(?p)(?q)(?r)$ are both q-labels.

The $\star$ operator extends conjunction to accommodate q-labels. Intuitively, if the constraint C_1 is labeled by p, and C_2 is labeled by $\neg p$, then both C_1 and C_2 must hold as long as the value of p is unknown, represented by $p \star \neg p \ = \ ?p$.

Definition 2 ($\star$). The operator, $\star\colon \mathcal{Q}^* \times \mathcal{Q}^* \rightarrow \mathcal{Q}^\star$, is defined thusly. First, for any $p \in \mathcal{P}$, $p \star p = p$ and $\neg p \star \neg p = \neg p$. Next, for any $p_1, p_2 \in \{p, \neg p, ?p\}$ for which $p_1 \neq p_2$, $p_1 \star p_2 = ?p$. Finally, for any q-labels $\ell_1, \ell_2 \in \mathcal{Q}^*$, $\ell_1 \star \ell_2 \in \mathcal{Q}^*$ denotes the conjunction obtained by applying $\star$ in pairwise fashion to matching literals from ℓ_1 and ℓ_2, and conjoining any unmatched literals.

Rule	Edge Generation	Conditions
LP_Z	$X \xrightarrow{\langle u, \alpha\rangle} W \xrightarrow{\langle v, \beta\rangle} \mathrm{Z}$ $X \xrightarrow{\langle u+v, \alpha\beta\rangle} \mathrm{Z}$	$u + v < 0$ and $\alpha\beta \in \mathcal{P}^*$
qR_0	$P? \xrightarrow[\langle w, \alpha\rangle]{\langle w, \alpha\tilde{p}\rangle} \mathrm{Z}$	$w < 0$, $\tilde{\pm}p \notin \alpha \in \mathcal{Q}^*$
qR_3^*	$P? \xrightarrow{\langle w, \alpha\rangle} \mathrm{Z} \xleftarrow[\langle \max\{v, w\}, \alpha \star \beta\rangle]{\langle v, \beta\tilde{p}\rangle} Y$	$w < 0$, $\tilde{\pm}p \notin \alpha \star \beta \in \mathcal{Q}^*$

$W, X, Y \in \mathcal{T}$; $\mathrm{Z} = 0$; $P? \in \mathcal{OT}$; and $u, v, w \in \mathbb{R}$.

Table 1: Edge-generation rules used by the $\pi\texttt{DC}_{18}$ algorithm

For example: $(p\neg q(?r)t) \star (qr\neg s) = p(?q)(?r)\neg st$.

The $\pi\texttt{DC}_{18}$ algorithm uses the three constraint-propagation/edge-generation rules shown in Table 1. Note that each rule only generates edges terminating at the zero time-point Z. For the qR_0 and qR_3^* rules, $\tilde{p} \in \{p, \neg p, ?p\}$, and $\tilde{\pm}p \notin \alpha$ means that none of p, $\neg p$ and $?p$ appear in α.

The LP_Z rule extends ordinary constraint-propagation in STNs to accommodate propositional labels. The label on the generated edge (shaded) is the conjunction of the labels on the pre-existing edges. The qR_0 rule applies when an observation time-point $P?$ has a lower bound that is conditioned by some propositional label. This rule stipulates that the condition on that lower bound cannot depend on the as-yet-unobserved value of the corresponding letter p. The qR_0 rule removes any occurrence of p from the propositional label. The qR_3^* rule similarly removes occurrences of p, but from a propositional label on a different edge. This rule can generate q-labeled edges: for example, if $\alpha = q$ and $\beta = \neg q$, then $\alpha \star \beta = ?q$.

Although the $\pi\texttt{DC}_{18}$ algorithm is sound and complete for π-DC checking, it can get bogged down cycling through negative q-loops. For example, recall the CSTN from the bottom of Fig. 1, a portion of which is shown in Fig. 2. It shows ten applications of the LP_Z, qR_0 and qR_3^* rules, generating the dashed edges in the order indicated by the parenthesized numbers, the end result of which is that the weights on the edges from $P?$ to Z, and $Q?$ to Z have changed from -13 to -15. After cycling through these interacting negative q-loops *several hundred more times,* the resulting edges will combine with the upper-bound edges from Z to $P?$ and Z to $Q?$ (not shown in Fig. 2) to generate negative loops with *consistent* labels, at which point the algorithm will correctly conclude that the network is *not* DC. However, although the CSTN at the *top* of Fig. 1 has a similar structure, the presence of $W?$ and constraints labeled by w and $\neg w$ combine to ensure that it *is* DC, which the $\pi\texttt{DC}_{18}$ algorithm will discover after cycling through the negative q-loops hundreds of times.

Rule	Edge Generation	Conditions
qInf	$\langle -\infty, \alpha \star \beta\rangle \circlearrowleft X \underset{\langle v, \beta\rangle}{\overset{\langle u, \alpha\rangle}{\rightleftarrows}} W$	$u < 0$, $u + v < 0$, and $\alpha \star \beta \in \mathcal{Q}^* \backslash \mathcal{P}^*$
qLP_1^+	$X \xrightarrow{\langle \overline{u}, \alpha\rangle} W \xrightarrow{\langle \overline{v}, \beta\rangle} Y$ $X \xrightarrow{\langle \overline{u} + \overline{v}, \alpha \star \beta\rangle} Y$	$\overline{u} + \overline{v} < 0$ and $[(\alpha \star \beta = \alpha\beta \in \mathcal{P}^*)$ or $(\overline{u} < 0)]$
qR_0^+	$P? \xrightarrow[\langle \overline{w}, \alpha\rangle]{\langle \overline{w}, \alpha\tilde{p}\rangle} X$	$\overline{w} < 0$, $\tilde{\pm}p \notin \alpha \in \mathcal{Q}^*$
qR_3^{*+}	$P? \xrightarrow{\langle \overline{w}, \alpha\rangle} X \xleftarrow[\langle \max\{\overline{v}, \overline{w}\}, \alpha \star \beta\rangle]{\langle \overline{v}, \beta\tilde{p}\rangle} Y$	$\overline{w} < 0$,$\tilde{\pm}p \notin \alpha \star \beta \in \mathcal{Q}^*$

$W, X, Y \in \mathcal{T}$; $u, v \in \mathbb{R}$; $\overline{u}, \overline{v}, \overline{w} \in [-\infty, \infty)$; and $P? \in \mathcal{OT}$.

Table 2: Edge-generation rules used by the $\pi\texttt{DC}^{\infty}_{19}$ algorithm

The $\pi\texttt{DC}^{\infty}_{19}$ Algorithm

Aiming to speed up π-DC checking by dealing more effectively with negative q-loops, Hunsberger and Posenato (2019) introduced a new set of edge-generation rules which, in this paper, we call the $\pi\texttt{DC}^{\infty}_{19}$ algorithm. It begins with the *qInf* rule shown in Table 2, that covers a special case of labeled propagation in which the two edges (from X to W to X) form a negative q-loop. (If $\alpha \star \beta = \alpha\beta \in \mathcal{P}^*$, then the CSTN can be immediately rejected as not DC.) They showed that instead of setting the weight on the generated loop to $u + v < 0$, it is sound to set it to $-\infty$. Intuitively, such a loop can be understood as saying that X cannot be executed as long as the label $\alpha \star \beta$ is (or might yet be) true. For example, a loop from X to X labeled by $\langle -\infty, (?p)q\rangle$ represents that X cannot be executed as long as p is unknown and q is (or might yet be) true. They showed that this rule can greatly speed up π-DC checking because instead of repeatedly cycling through negative q-loops many times, their algorithm may cycle through them only once, using the rest of the rules from Table 2, which are straightforward extensions to the rules from Table 1 to accommodate $-\infty$ and to generate edges pointing at any node—not just Z. In addition, the qLP_1^+ rule can generate q-labeled edges and the qInf rule can be

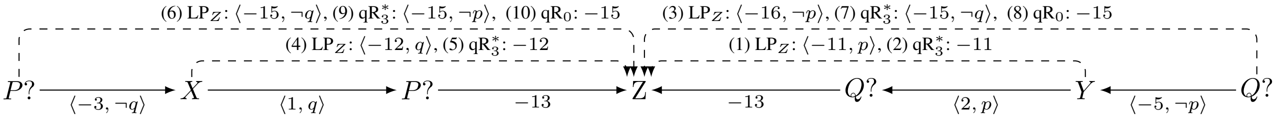


Figure 2: The $\pi\texttt{DC}_{18}$ algorithm cycling through a pair of interacting negative q-loops

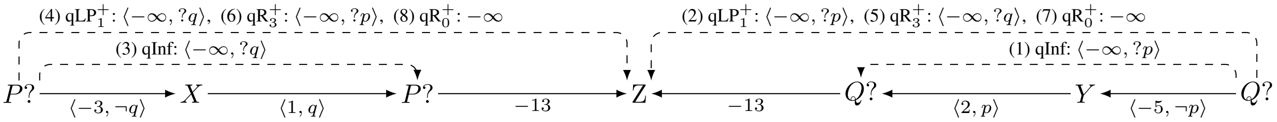


Figure 3: The $\pi\texttt{DC}^{\infty}_{19}$ algorithm more efficiently handling the interacting negative q-loops from Fig. 2

incorporated into the qLP_1^+ rule as a post-process.

Fig. 3 shows that the rules from Table 2 only pass through the negative q-loops from Fig. 2 *once* to generate lower bounds of ∞ for $P?$ and $Q?$. Since $P?$ and $Q?$ have upper bounds of 50 (cf. Fig. 1), the CSTN must be non-DC. Similar results apply to the DC network from the top of Fig. 1.

Although expected to outperform the $\pi\texttt{DC}_{18}$ algorithm on networks with negative q-loops, Hunsberger and Posenato (2019) did not empirically evaluate the $\pi\texttt{DC}_{19}^{\infty}$ algorithm. (Their only intent was to show its usefulness in a context where weights on edges could be piecewise-linear functions.)

A Faster π-DC-Checking Algorithm

This section introduces a new π-DC-checking algorithm for CSTNs, called $\pi\texttt{DC}_{20}^{\infty}$, that builds on the algorithms seen above. The primary insight is that the semantics of satisfying an edge labeled by $\langle -\infty, \alpha\rangle$, for some $\alpha \in \mathcal{Q}^*\backslash\mathcal{P}^*$ does not depend on the target node of the edge, but only on its source node. As a result, much of the propagation of such labeled values by the $\pi\texttt{DC}_{19}^{\infty}$ algorithm is redundant. The $\pi\texttt{DC}_{20}^{\infty}$ algorithm avoids this problem by associating such labeled values only with *nodes,* not edges.[1] The $\pi\texttt{DC}_{20}^{\infty}$ algorithm also separates the job of finding negative q-loops, which it does in a pre-processing phase, from the main algorithm.

The Semantics of Constraints on Nodes

Hunsberger and Posenato (2018) defined the semantics of satisfying a (lower-bound) q-labeled constraint $X \xrightarrow{\langle\delta,\alpha\rangle} Z$ for any $\delta < 0$ and $\alpha \in \mathcal{Q}^*$. Applying this definition to cases where $\delta = -\infty$, and letting the target node be any $Y \in \mathcal{T}$, yields the following (Hunsberger and Posenato 2019).

Definition 3. The execution strategy σ satisfies the q-labeled constraint $X \xrightarrow{\langle -\infty,\alpha\rangle} Y$ if for each scenario s:

(1) $[\sigma(s)]_X \geq [\sigma(s)]_Y + \infty$; or

(2) some $\tilde{a} \in \{a, \neg a, ?a\}$ appears in α such that $\sigma(s)$ observes a π-before Y and $s \not\models \tilde{a}$.[2]

Since clause (1) cannot be satisfed, it follows that σ can only satisfy such a constraint if $\sigma(s)$ does not execute X until it first executes some observation time-point $A?$ that generates a value for a that ensures that $s \not\models \alpha$. The critical point is that such a constraint only applies to the source node X; it does not involve Y at all. For this reason, it makes sense to associate such a constraint to the *node* X, not to the *edge* from X to Y. Furthermore, it is pointless to forward-propagate such constraints, because the resulting edge would have the same source node, and hence would be redundant.

Algorithm 1: $\texttt{NQLFinder}(G)$

```
Input: CSTN $G = (\mathcal{T}, E)$.
Output: $G$ modified by NQLF rule.
$Q := E,\ newQ := \{\},\ n := |\mathcal{T}| - 1$
while $Q \neq \emptyset$ and $n > 0$ do
    while $Q \neq \emptyset$ do
        $(X, Y)$ := extract an edge from $Q$
        foreach $(Y, W) \in E$ do
            if $(X, W) \in E$ and $n \neq |\mathcal{T}| - 1$ then
                continue // Update $(X, W)$ only once
            eXWfilled := NQLF($(X, Y), (Y, W)$)
            if eXWfilled is new or modified then
                $newQ = newQ \cup \{eXWfilled\}$
    $n := n - 1$
    $Q = newQ$
```

Rule	Edge Generation	Conditions
qLP	$\begin{tikzcd} X \arrow[r, "\langle u,\alpha\rangle"] \arrow[rr, bend right, "\langle u+v,\alpha\star\beta\rangle"'] & W \arrow[r, "\langle v,\beta\rangle"] & Y \end{tikzcd}$	$(u < 0$ and $u + v < 0)$ or $(\alpha \star \beta \in \mathcal{P}^*)$
1	$X \underset{\langle v,\beta\rangle}{\overset{\langle u,\alpha\rangle}{\rightleftarrows}} W$ adds $X_{\langle -\infty,\alpha\star\beta\rangle}$	$u < 0$, $u + v < 0$, and $\alpha \star \beta \in \mathcal{Q}^*$
qR_0	$P? \xrightarrow[\langle w,\alpha\rangle]{\langle w,\alpha\tilde{p}\rangle}$ X	$w < 0$, $\tilde{\pm}p \notin \alpha \in \mathcal{Q}^*$
2	$P?_{\langle -\infty,\tilde{p}\alpha\rangle}$ adds $P?_{\langle -\infty,\alpha\rangle}$	$\tilde{\pm}p \notin \alpha \in \mathcal{Q}^*$

$W, X, Y \in \mathcal{T}$; $P? \in \mathcal{OT}$; $u, v, w \in \mathbb{R}$; In **1**, if $\alpha \star \beta \in \mathcal{P}^*$, then the network must be non-DC.

Table 3: Edge-generation rules for `NQLFinder`

Finding Negative Q-Loops

The `NQLFinder` algorithm, shown in Algorithm 1, is a preprocess that uses a single, composite rule called NQLF that combines the rules listed in Table 3 to find all negative q-loops having at most $n = |\mathcal{T}|$ time-points.[3] The qLP rule propagates forward from each source node X, generating negative-length edges, but note that v (i.e., the length of the second edge in the rule) may be non-negative. The **1** rule is similar to the qInf rule from Table 2, except that it generates a labeled value associated with the *node* X, not the *edge* from X to X. The qR_0 and **2** rules are used as post-processes to qLP and **1**, respectively, to remove instances of any $\tilde{p} \in \{p, \neg p, ?p\}$ when X is the corresponding observation time-point $P?$. In the implementation, the four rules from Table 3 are folded into a single composite rule, called NQLF.

---

[1] Whereas the propositional labels, $\alpha \in \mathcal{P}^*$, that Tsamardinos et al. (2003) applied to nodes in (unstreamlined) CSTNs specified the scenarios in which nodes must be executed, our application of $\langle -\infty, \alpha\rangle$, with $\alpha \in \mathcal{Q}^*\backslash\mathcal{P}^*$, to a node specifies a *dynamic* constraint on when that node can be executed. Completely different.

[2] The π-before relation (in this case) stipulates that in scenario s, σ executes $A?$ before Y, or simultaneous with Y, but ordered before Y. (See the definitions in the Appendix.) For convenience, this definition assumes the convention that $s \not\models ?p$ for any $p \in \mathcal{P}^*$.

[3] A negative q-loop with more than n time-points must have a sub-loop that is a negative q-loop with at most n time-points.

Rule	Edge Generation	Conditions
$\mathrm{qLP}_{\mathrm{Z}}$	$\underbrace{X \xrightarrow{\langle u, \alpha\rangle} W \xrightarrow{\langle v, \beta\rangle} \mathrm{Z}}_{\langle u+v, \alpha \star \beta\rangle}$	$(\alpha \star \beta = \alpha\beta \in \mathcal{P}^*)$ or $(u < 0$ and $u + v < 0)$.
3	$X \xrightarrow{\langle u, \alpha\rangle} W_{\langle\text{-}\infty, \beta\rangle}$ adds $X_{\langle\text{-}\infty, \alpha\star\beta\rangle}$	$u < 0, \alpha\star\beta \in \mathcal{Q}^* \setminus \mathcal{P}^*$
qR_0	$P? \xrightarrow[\langle w, \alpha\rangle]{\langle w, \alpha\tilde{p}\rangle} \mathrm{X}$	$w < 0, \tilde{\pm}p \notin \alpha \in \mathcal{Q}^*$
2	$P?_{\langle\text{-}\infty, \tilde{p}\alpha\rangle}$ adds $P?_{\langle\text{-}\infty, \alpha\rangle}$	$\tilde{\pm}p \notin \alpha$
qR^*_3	$P? \xrightarrow{\langle w, \alpha\rangle} \mathrm{Z} \xleftarrow[\langle \max\{v, w\}, \alpha \star \beta\rangle]{\langle v, \beta\tilde{p}\rangle} Y$	$w < 0, \tilde{\pm}p \notin \alpha\star\beta \in \mathcal{Q}^*$
$\mathbf{5}_{\mathrm{Z}}$	$P? \xrightarrow{\langle u, \alpha\rangle} Z \xleftarrow[\langle u, \alpha \star \beta\rangle]{} Y_{\langle\text{-}\infty, \tilde{p}\beta\rangle}$	$u < 0$
$\mathbf{4.1}_{\mathrm{Z}}$	$P?_{\langle\text{-}\infty, \alpha\rangle} \qquad \mathrm{Z} \xleftarrow[\langle u, \alpha \star \beta\rangle]{\langle u, \tilde{p}\beta\rangle} Y$	$u < 0$
4.2	$P?_{\langle\text{-}\infty, \alpha\rangle}\ Y_{\langle\text{-}\infty, \tilde{p}\beta\rangle}$ adds $Y_{\langle\text{-}\infty, \alpha\star\beta\rangle}$	
$W, X, Y \in \mathcal{T}$; $\mathrm{Z} = 0$; $P? \in \mathcal{OT}$; $u, v, w \in \mathbb{R}$. In **3**, if $\alpha \star \beta \in \mathcal{P}^*$, then network must be non-DC.		

Table 4: Edge-generation rules for the $\pi\mathtt{DC}^{\infty}_{20}$ algorithm

The overall aim of the `NQLFinder` algorithm is to find all nodes that can be labeled by $\langle -\infty, \alpha\rangle$ for some α. (A single node may have a set of such labels, each with a different α.) Often, not every node in a negative q-loop can be so labeled (e.g., source nodes of non-negative-length edges). When done, any edges discovered by the `NQLFinder` algorithm are discarded; only the node-constraints are kept.

When `NQLFinder` is run on the CSTN at the bottom of Fig. 1, single applications of the **1** rule generate labels of $\langle -\infty, ?p\rangle$ for $Q?$, and $\langle -\infty, ?q\rangle$ for $P?$. Afterward, the main algorithm, discussed below, can use rules **4.2** and **2** from Table 4 to generate the unconditional lower bounds of ∞ on $P?$ and $Q?$ which, given their finite upper bounds, implies that the network is non-DC.

Propagating Constraints

The main part of the $\pi\mathtt{DC}^{\infty}_{20}$ algorithm uses the rules shown in Table 4. Like the $\pi\mathtt{DC}_{18}$ rules from Table 1, all edges generated by the $\pi\mathtt{DC}^{\infty}_{20}$ rules have Z as their target, and have *finite* numerical weights. Like the $\pi\mathtt{DC}^{\infty}_{19}$ rules from Table 2, the $\pi\mathtt{DC}^{\infty}_{20}$ rules generate labels such as $\langle -\infty, \alpha\rangle$; however, such labels are applied to *nodes,* not edges. The qLP_Z, qR_0 and qR^*_3 rules are identical to those used by the $\pi\mathtt{DC}_{18}$ algorithm, except that the qLP_Z rule accommodates q-labels.[4] Each instance of the **3** rule propagates a $\langle -\infty, \beta\rangle$ label on a node backward across an edge to generate a new node label. The **2** rule is the same as the one used by `NQLFinder` (cf. Table 3). The $\mathbf{5}_{\mathrm{Z}}$, $\mathbf{4.1}_{\mathrm{Z}}$, and **4.2** rules extend the qR^*_3 rule to accommodate $\langle -\infty, \alpha\rangle$ labels on nodes in different positions.

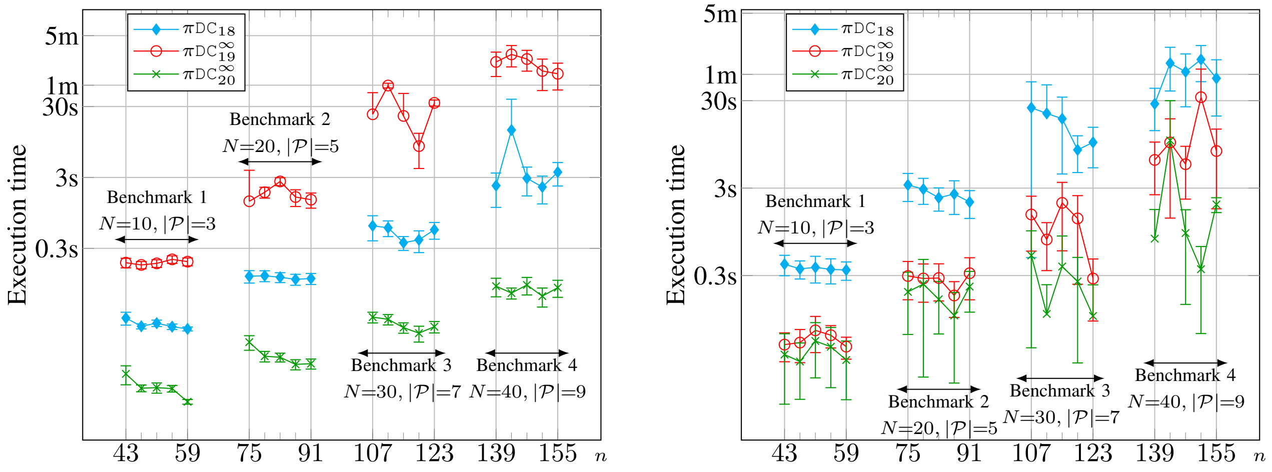
Algorithm 2: $\pi\mathtt{DC}^{\infty}_{20}$ (G)

```
Input: CSTN $G = (\mathcal{T}, E)$
Output: Consistency status: YES/NO
$\mathrm{Z}.d := \{\langle 0, \boxdot\rangle\}$      // Z = first node; $v.d$ is $v$'s potential
NQLFinder($G$)                 // Generate $\langle -\infty, \alpha\rangle$ values
$Q := \{\mathrm{Z}\}$
while $Q \neq \emptyset$ do
    $QObs := \{\}$
    while $Q \neq \emptyset$ do // Update node distances
        $X :=$ extract a node from $Q$
        foreach $eYX := (Y, X) \in E$ do
            foreach $\langle u, \alpha\rangle \in X.d$ do
                foreach $\langle v, \beta\rangle \in eYX$ do
                    potLP($\langle u, \alpha\rangle, \langle v, \beta\rangle$)
            if $Y.d$ potential was updated then
                Insert $Y$ in $Q$
                if $Y$ is an observation time point then
                    Insert $Y$ in $QObs$
    // Apply potQR$^*_3$ among obs. time-points ONLY
    $QObs1 = QObs$
    while $QObs1 \neq \emptyset$ do
        $A? :=$ extract a node from $QObs1$
        foreach observation time-point $X? \in V$ do
            // Apply potqR$^*_3$ to $X?$ w.r.t $A?$
            foreach $\langle u, \tilde{a}\gamma\rangle \in X.d$ do
                if $\gamma \in P^*$ and $u = -\infty$ then return NO
                $X.d(\gamma) := u$
            if $X.d$ potential was updated then
                Insert $X$ in $Q$, $QObs$ and in $QObs1$
    // Apply potQR$^*_3$ to other time-points
    foreach observation time point $A? \in QObs$ do
        foreach $X \in V$ do
            if $X$ is an obs. time-point then continue
            // Apply potqR$^*_3$ to $X$ w.r.t $A?$
            foreach $\langle u, \tilde{a}\gamma\rangle \in X.d$ do
                if $\gamma \in P^*$ and $u = -\infty$ then return NO
                $X.d(\gamma) := u$
            if $X.d$ potential was updated then
                Insert $X$ in $Q$
return YES
```

Since all edges manipulated by the $\pi\mathtt{DC}^{\infty}_{20}$ algorithm have Z as their target, and the only other labeled values are associated with nodes, our implementation of the $\pi\mathtt{DC}^{\infty}_{20}$ algorithm, shown as Algorithm 2, makes the following unifying simplification. If an edge from X to Z has a labeled value $\langle \delta, \alpha\rangle$,then the implementation treats that labeled value as a *labeled potential* that it stores with the node X. Because labeled values on edges from X to Z only have *finite* weights, such labeled potentials are easily distinguished from the labeled values $\langle -\infty, \alpha\rangle$ that the `NQLFinder` algorithm assigns to nodes. Our implementation also treats these labeled values as labeled potentials associated with nodes. As a result, our implementation deals only with labeled potentials of nodes; it does not deal with edge constraints at all. Thus, the qLP_Z and **3** rules can be combined into one rule, to which the qR_0 and **2** rules can be appended as a post-process, resulting in a single composite rule called potLP in Algorithm 2. Similarly, the qR^*_3, $\mathbf{5}_Z$, $\mathbf{4.1}_Z$**,** and **4.2** rules can be combined into a single composite rule called potQR^*_3 in Algorithm 2.

---

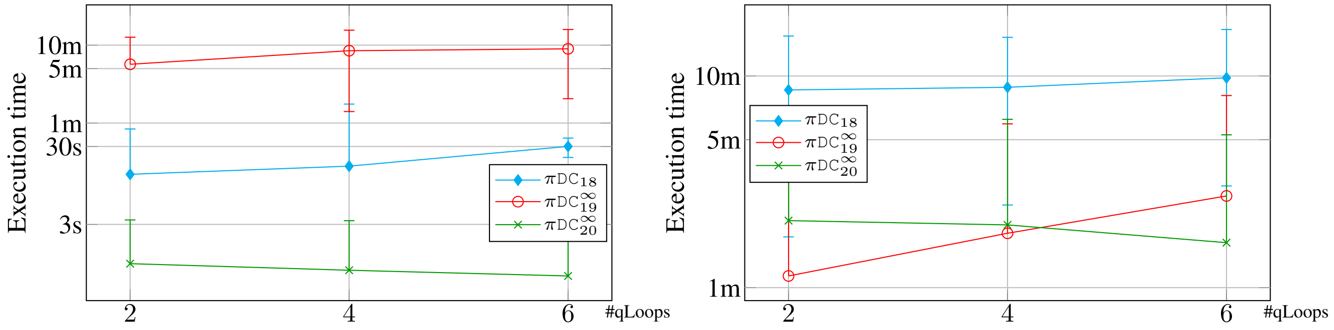
[4] Since these rules are more general than their counterparts in the $\pi\mathtt{DC}_{18}$ algorithm, the $\pi\mathtt{DC}^{\infty}_{20}$ algorithm is necessarily complete.



(a) **Benchmarks with DC Instances**

(b) **Benchmarks with non-DC Instances**.

Figure 4: Execution times vs number of nodes



(a) **Benchmarks with DC Instances**

(b) **Benchmarks with non-DC Instances**.

Figure 5: Results of `100n7pQL6nQL1pQL` benchmark

In summary, unlike all previous algorithms, Algorithm 2 does not add any edges to the network and checks the dynamic consistency by determining the minimal distance to Z for each node in relevant scenarios. This approach avoids a large amount of redundant propagation of labeled values on edges that is done by other algorithms.

Experimental Evaluation

This section compares the performance of our new $\pi\mathtt{DC}^{\infty}_{20}$ algorithm against the existing $\pi\mathtt{DC}_{18}$ and $\pi\mathtt{DC}^{\infty}_{19}$ algorithms. $\pi\mathtt{DC}^{\infty}_{20}$ refers to our implementation of Algorithm 2; $\pi\mathtt{DC}_{18}$ is the freely available implementation (Posenato 2018) of the π-DC-checking algorithm of Hunsberger and Posenato (2018); $\pi\mathtt{DC}^{\infty}_{19}$ is our implementation of the alternative π-DC-checking algorithm proposed by Hunsberger and Posenato (2019). All algorithms and procedures were implemented in Java and executed on a JVM 8 in a Linux box with two AMD Opteron 4334 CPUs and 64GB of RAM.

All implementations were tested on instances of the four benchmarks from Hunsberger and Posenato (2016). Each benchmark has at least 250 DC and 250 non-DC CSTNs, obtained from random workflow schemata generated by the ATAPIS toolset. The numbers of activities (N) of random workflows and choice connectors (corresponding to CSTN observations ($|\mathcal{P}|$)) were varied, as shown in Fig. 4.

We fixed a time-out of 10 minutes (m) for the execution of each algorithm on each instance. For the DC instances, $\pi\mathtt{DC}^{\infty}_{19}$ timed out on 32 of the 250 instances, while $\pi\mathtt{DC}_{18}$ timed out on only 3. Most of time-outs occurred in benchmark 4. For the not-DC instances, $\pi\mathtt{DC}^{\infty}_{19}$ timed out on 2 of the 250 instances, while $\pi\mathtt{DC}_{18}$ timed out on 18. The $\pi\mathtt{DC}^{\infty}_{20}$ algorithm never timed out.

Fig. 4 displays the average execution times of the three algorithms across all eight benchmarks (4 for DC instances, 4 for non-DC instances), each point representing the average execution time for instances of the given size. The size of the benchmarks allows the determination of 95% confidence intervals for the results. The results demonstrate that the previous algorithms $\pi\mathtt{DC}_{18}$ and $\pi\mathtt{DC}^{\infty}_{19}$ perform differently for different kinds of networks: $\pi\mathtt{DC}_{18}$ is better than $\pi\mathtt{DC}^{\infty}_{19}$ when instances are DC, while $\pi\mathtt{DC}^{\infty}_{19}$ is better when instances are not DC. The reason is that $\pi\mathtt{DC}_{18}$ generates labeled values only on edges pointing at Z, while $\pi\mathtt{DC}^{\infty}_{19}$ can generate labeled values for any edge. Therefore, when instances are DC (i.e., no negative cycle with a consistent label), the propagations are exhausted earlier by the $\pi\texttt{DC}_{18}$ algorithm. In contrast, when an instance is non-DC, $\pi\texttt{DC}^{\infty}_{19}$ tends to detect the negative loop with a consistent label much more quickly, due to its more efficient processing of negative q-loops. (The $\pi\texttt{DC}_{18}$ algorithm can cycle repeatedly through negative q-loops until some upper bound is violated, which can take a long time if the upper bound is relatively large.)

The $\pi\texttt{DC}^{\infty}_{20}$ algorithm can be viewed as combining the strengths of the $\pi\texttt{DC}_{18}$ and $\pi\texttt{DC}^{\infty}_{19}$ algorithms. First, it identifies negative q-loops more efficiently as a pre-process. Second it uniformly treats all constraints as labeled potentials on nodes, avoiding the redundant propagations of $\langle -\infty, \alpha\rangle$ values by $\pi\texttt{DC}^{\infty}_{19}$. Since it updates only the potentials of nodes, when an instance is DC, it updates such potentials similarly to how the $\pi\texttt{DC}_{18}$ algorithm updates edges pointing at Z, without any other useless computations. When an instance is non-DC, the `NQLFinder` pre-process can detect negative q-loops efficiently, and the main $\pi\texttt{DC}^{\infty}_{20}$ algorithm can manage the node potentials more efficiently than $\pi\texttt{DC}^{\infty}_{19}$. Therefore, its performance is better than both of the other algorithms when instances are positive, while it is more or less equivalent to the $\pi\texttt{DC}^{\infty}_{19}$ algorithm when instances are negative.

To study the behavior of the three algorithms with respect to the structure of possible CSTN instances, we set up a new random generator of CSTN instances by which it is possible to generate random instances having many specific features. Some features can be given as input to the random generator: number of nodes, number of propositions, probability of an edge for each pair of nodes, minimal number of negative q-loops, number of propositions used for generating q-loops, number of edges in a negative q-loops, the value (circuit weight) of negative q-loops, minimal and maximal edge weight, number of observation time-points in q-loops, minimal distance from observation time-points to Z, etc. Then, we built two new benchmarks.

The first benchmark, `100n7pQL6nQL1pQL`, contains 300 random instances (150 DC, 150 not-DC) each of which contains 100 nodes, 7 propositions, and some negative q-loops having 6 edges, cycle weight -1, and each containing just 1 proposition. The benchmark is divided into 3 sub-benchmarks of 50 instances each: the first contains instances in which at least 2 negative q-loops are present, the second contains instances having at least 4 negative q-loops, and the third contains instances having at least 6 negative q-loops. In each instance, the weight of an edge is a random value in the range $[-150, 150]$. Figure 5 depicts the execution times of the three algorithms on the `100n7pQL6nQL1pQL` benchmark. The time-out was fixed to 15 m.

For DC instances, $\pi\texttt{DC}^{\infty}_{19}$ timed out on approx. 43% of the instances, while $\pi\texttt{DC}_{18}$ and $\pi\texttt{DC}^{\infty}_{20}$ never timed out. For non-DC instances, $\pi\texttt{DC}^{\infty}_{19}$ timed out on approx. 6% of the 150 instances, $\pi\texttt{DC}_{18}$ for approx. 51%, and $\pi\texttt{DC}^{\infty}_{20}$ for approx. 1%. These results confirm that instances containing negative q-loops are harder to solve than those without negative q-loops because the numbers of time-outs increased. Then, the $\pi\texttt{DC}^{\infty}_{20}$ algorithm is still the preferred algorithm for testing both DC and non-DC instances. Moreover, the results suggest that the difference among having 2, 4, and 6 negative q-loops does not significantly affect the execution times for any of the algorithms.

The second benchmark, `100n7pQL6nQL1pQLFarObs`, contains the same instances as the first benchmark, but with the distances of observation time-points from Z modified. Each observation time-point has an edge to Z with a random value (distance) in the range [-450, -300]. In this way, we wanted to study how the algorithms work for solving negative q-loops. Figure 6 depicts the execution times of the three algorithms on the `100n7pQL6nQL1pQLFarObs` benchmark. The time-out was fixed to 15 m. For the DC instances, $\pi\texttt{DC}^{\infty}_{19}$ timed out for approx. 36% of the 150 instances, while $\pi\texttt{DC}_{18}$ and $\pi\texttt{DC}^{\infty}_{20}$ never timed out. For the non-DC instances, $\pi\texttt{DC}^{\infty}_{19}$ timed out for approx. 8% of the 150 instances, while $\pi\texttt{DC}_{18}$ for approx. 59%, and $\pi\texttt{DC}^{\infty}_{20}$ for approx. 4%. Although we had expected the $\pi\texttt{DC}_{18}$ algorithm to perform much worse on the non-DC instances in this benchmark, the results did not confirm this. We will explore different benchmarks to further understand the different behaviors of the three algorithms.

The main takeaway from our empirical evaluation is that the new $\pi\texttt{DC}^{\infty}_{20}$ algorithm performs significantly better than the existing algorithms across many types of benchmarks, and always performs at least as well as those algorithms on all benchmarks.

Conclusions

This paper presented a new approach to DC checking for CSTNs that results in an algorithm that is empirically demonstrated to be significantly faster than existing DC-checking algorithms across not only existing benchmarks, but also across a new set of benchmarks. The algorithm more efficiently identifies important graphical structures called negative q-loops and more efficiently manages the propagation of labeled values of the form $\langle -\infty, \alpha\rangle$. In addition, unlike previous algorithms, the main phase of the new algorithm only updates labeled values—whether finite or infinite—on nodes, not edges.

For future work, we aim to evaluate the new algorithm across a wider variety of benchmark problems to determine which graphical features most significantly impact its performance.

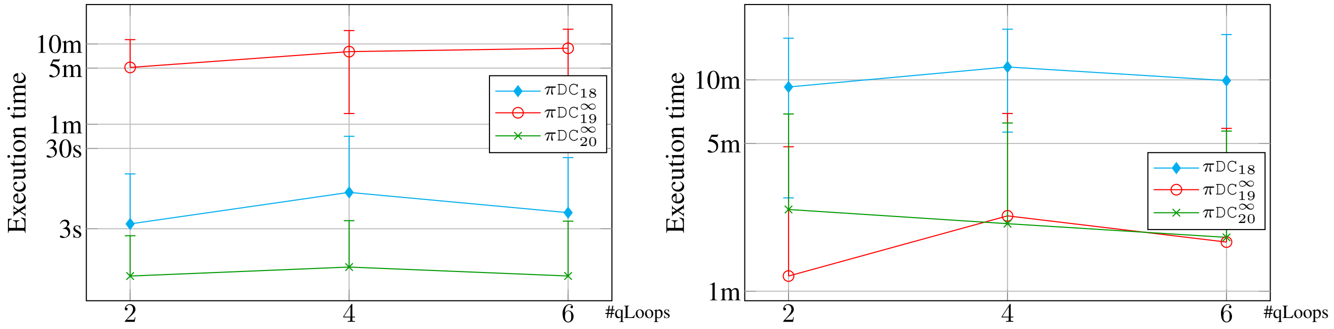
Appendix: Definition of π-DC for CSTNs

The definitions give below are expressed in the form used by Hunsberger and Posenato (2018).

Definition 4 (Labels)**.** Let $\mathcal{P}$ be a set of propositional letters. A *label* is a conjunction of (positive or negative) literals from $\mathcal{P}$. The empty label is notated $\boxdot$; and $\mathcal{P}^*$ denotes the set of all satisfiable labels with literals from $\mathcal{P}$.

Definition 5 (CSTN)**.** A *Conditional Simple Temporal Network* (CSTN) is a tuple, $\langle \mathcal{T}, \mathcal{P}, \mathcal{C}, \mathcal{OT}, \mathcal{O}\rangle$, where:

- $\mathcal{T}$ is a finite set of real-valued time-points (i.e., variables);
- $\mathcal{P}$ is a finite set of propositional letters (or propositions);
- $\mathcal{C}$ is a set of *labeled* constraints, each having the form, $(Y - X \leq \delta,\ \ell)$, where $X, Y \in \mathcal{T}$, $\delta \in \mathbb{R}$, and $\ell \in \mathcal{P}^*$;
- $\mathcal{OT} \subseteq \mathcal{T}$ is a set of observation time-points (OTPs); and
- $\mathcal{O}\colon \mathcal{P} \to \mathcal{OT}$ is a bijection that associates a unique observation time-point to each propositional letter.



(a) **Benchmarks with DC Instances**

(b) **Benchmarks with non-DC Instances**.

Figure 6: Results of `100n7pQL6nQL1pQLFarObs` benchmark where all observation nodes have a big distance from Z.

In a CSTN graph, the observation time-point for p (i.e., $\mathcal{O}(p)$) is usually denoted by $P?$; and each labeled constraint, $(Y - X \le \delta, \ell)$, is represented by an arrow from X to Y, annotated by the *labeled value* $\langle \delta, \ell \rangle$: $X \xrightarrow{\langle \delta, \ell \rangle} Y$. (If ℓ is empty, then the arrow is labeled by δ, as in an STN graph.) Since X and Y may participate in multiple constraints of the form, $(Y - X \le \delta_i, \ell_i)$, the edge from X to Y may have multiple labeled values of the form, $\langle \delta_i, \ell_i \rangle$.

Definition 6 (Schedule)**.** A *schedule* for a set of time-points $\mathcal{T}$ is a mapping, $\psi\colon \mathcal{T} \to \mathbb{R}$. The set of all schedules *for any subset of* $\mathcal{T}$ is denoted by Ψ.

Definition 7 (Scenario)**.** A function, $s\colon \mathcal{P} \to \{\top, \bot\}$, that assigns a truth value to each $p \in \mathcal{P}$ is called a *scenario*. For any label $\ell \in \mathcal{P}^*$, the truth value of ℓ determined by s is denoted by $s(\ell)$. $\mathcal{I}$ denotes the set of all scenarios over $\mathcal{P}$.

The projection of a CSTN onto a scenario, s, is the STN obtained by including only the constraints whose labels are true under s (i.e., that must be satisfied in that scenario).

Definition 8 (Projection)**.** Let $\mathcal{S} = \langle \mathcal{T}, \mathcal{P}, \mathcal{C}, \mathcal{OT}, \mathcal{O} \rangle$ be any CSTN, and s any scenario over $\mathcal{P}$. The *projection* of $\mathcal{S}$ onto s—notated $\mathcal{S}(s)$—is the STN, $(\mathcal{T}, \mathcal{C}^+_s)$, where:

$$\mathcal{C}^+_s = \{(Y - X \le \delta) \mid \exists \ell, (Y - X \le \delta, \ell) \in \mathcal{C} \wedge s(\ell) = \top\}$$

The truth values of propositions in a CSTN are not known in advance, but a π-*dynamic* execution strategy can *react instantaneously* to observations. To make instantaneous reactivity plausible, a π-execution strategy must specify an order of dependence among simultaneous observations.

Definition 9 (Order of dependence)**.** For any scenario s, and ordering $(P_1?, \ldots, P_k?)$ of observation time-points, where $k = |\mathcal{OT}|$, an *order of dependence* is a permutation π over $(1, 2, \ldots, k)$; and for each $P? \in \mathcal{OT}$, $\pi(P?) \in \{1, 2, ..., k\}$ denotes the integer position of $P?$ in that order. For any *non*-observation time-point X, we set $\pi(X) = \infty$. Finally, Π_k denotes the set of all permutations over $(1, 2, \ldots, k)$.

Definition 10 (π-Execution Strategy)**.** Given any CSTN $\mathcal{S} = \langle \mathcal{T}, \mathcal{P}, \mathcal{C}, \mathcal{OT}, \mathcal{O} \rangle$, let $k = |\mathcal{OT}|$. A π-*execution strategy* for $\mathcal{S}$ is a mapping, $\sigma\colon \mathcal{I} \to (\Psi \times \Pi_k)$, such that for each scenario s, $\sigma(s)$ is a pair (ψ, π) where $\psi\colon \mathcal{T} \to \mathbb{R}$ is a schedule and $\pi \in \Pi_k$ is an order of dependence. For any $X \in \mathcal{T}$, $[\sigma(s)]_X$ denotes the execution time of X (i.e., $\psi(X)$); and for any $P? \in \mathcal{OT}$, $[\sigma(s)]^\pi_{P?}$ denotes the position of $P?$ in the order of dependence (i.e., $\pi(P?)$). Finally, a π-dynamic strategy must be *coherent:* for any scenario s, and any $P?, Q? \in \mathcal{OT}$, $[\sigma(s)]_{P?} < [\sigma(s)]_{Q?}$ implies $[\sigma(s)]^\pi_{P?} < [\sigma(s)]^\pi_{Q?}$ (i.e., if $\sigma(s)$ *schedules* $P?$ before $Q?$, then it *orders* $P?$ before $Q?$).

Definition 11 (Viability)**.** The π-execution strategy σ is called *viable* for the CSTN $\mathcal{S}$ if for each scenario s, the schedule ψ is a solution to the projection $\mathcal{S}(s)$, where $\sigma(s) = (\psi, \pi)$.

Definition 12 (π-History)**.** Let σ be any π-execution strategy for some CSTN $\mathcal{S} = \langle \mathcal{T}, \mathcal{P}, \mathcal{C}, \mathcal{OT}, \mathcal{O} \rangle$, s any scenario, t any real number, and $d \in \{1, 2, \ldots, |\mathcal{OT}|\} \cup \{\infty\}$ any integer position (or infinity). The π-*history* of (t, d) for the scenario s and strategy σ—denoted by $\pi\mathit{Hist}(t, d, s, \sigma)$—is the set

$$\{(p, s(p)) \mid P? \in \mathcal{OT},\ [\sigma(s)]_{P?} \le t, \pi(P?) < d\}.$$

The π-history of (t, d) specifies the truth value of each $p \in \mathcal{P}$ that is observed *before* t, or *at* t if the corresponding $P?$ is ordered *before* position d by the permutation π.

Definition 13 (π-Dynamic Strategy)**.** A π-execution strategy, σ, for a CSTN is called π-*dynamic* if for every pair of scenarios, s_1 and s_2, and every time-point $X \in \mathcal{T}$:

let: $t = [\sigma(s_1)]_X$, and $d = [\sigma(s_1)]^\pi_X$.
if: $\pi\mathit{Hist}(t, d, s_1, \sigma) = \pi\mathit{Hist}(t, d, s_2, \sigma)$
then: $[\sigma(s_2)]_X = t$ and $[\sigma(s_2)]^\pi_X = d$.

Thus, if σ executes X at time t and position d in scenario s_1, and the histories, $\pi\mathit{Hist}(t, d, s_1, \sigma)$ and $\pi\mathit{Hist}(t, d, s_2, \sigma)$, are the same, then σ must also execute X at time t and in position d in s_2. (X may be an observation time-point.)

Definition 14 (π-Dynamic Consistency)**.** A CSTN, $\mathcal{S}$, is π-*dynamically consistent* (π-DC) if there exists a π-execution strategy for $\mathcal{S}$ that is both viable and π-dynamic.

References

Cairo, M.; Hunsberger, L.; Posenato, R.; and Rizzi, R. 2017. A Streamlined Model of Conditional Simple Temporal Networks - Semantics and Equivalence Results. In *24th Int. Symp. on Temporal Representation and Reasoning (TIME-2017)*, volume 90 of *LIPIcs*, 10:1–10:19.

Cairo, M.; Comin, C.; and Rizzi, R. 2016. Instantaneous Reaction-Time in Dynamic-Consistency Checking of Conditional Simple Temporal Networks. In *23rd Int. Symp. on Temporal Representation and Reasoning (TIME-2016)*, 80–89.

Comin, C., and Rizzi, R. 2015. Dynamic consistency of conditional simple temporal networks via mean payoff games: a singly-exponential time dc-checking. In *22st Int. Symp. on Temporal Representation and Reasoning (TIME 2015)*, 19–28.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal Constraint Networks. *Artificial Intelligence* 49(1-3):61–95.

Hunsberger, L., and Posenato, R. 2016. Checking the dynamic consistency of conditional temporal networks with bounded reaction times. In Coles, A. J.; Coles, A.; Edelkamp, S.; Magazzeni, D.; and Sanner, S., eds., *26th Int. Conf. on Automated Planning and Scheduling, ICAPS 2016*, 175–183.

Hunsberger, L., and Posenato, R. 2018. Simpler and Faster Algorithm for Checking the Dynamic Consistency of Conditional Simple Temporal Networks. In *26th Int. Joint Conf. on Artificial Intelligence, (IJCAI-2018)*, 1324–1330.

Hunsberger, L., and Posenato, R. 2019. Propagating Piecewise-Linear Weights in Temporal Networks. In *29th International Conference on Automated Planning and Scheduling, ICAPS 2019*, volume 29, 223–231. AAAI Press.

Hunsberger, L.; Posenato, R.; and Combi, C. 2012. The Dynamic Controllability of Conditional STNs with Uncertainty. In *Work. on Planning and Plan Execution for Real-World Systems (PlanEx) at ICAPS-2012*, 1–8.

Posenato, R. 2018. The CSTNU toolset. version 1.23. http://profs.scienze.univr.it/∼posenato/software/cstnu.

Tsamardinos, I.; Vidal, T.; and Pollack, M. E. 2003. CTP: A new constraint-based formalism for conditional, temporal planning. *Constraints* 8:365–388.