Robust Execution of Probabilistic STNs

Luke Hunsberger
Vassar College, USA

Roberto Posenato
University of Verona, Italy

Abstract

A Probabilistic Simple Temporal Network (PSTN) is a formalism for representing and reasoning about actions subject to temporal constraints, where some action durations may be uncontrollable, modeled using continuous probability density functions. Recent work aims to manage this kind of uncertainty during execution by approximating a PSTN by a Simple Temporal Network with Uncertainty (STNU) (for which well-known execution strategies exist) and using an STNU execution strategy to execute the PSTN, hoping that its probabilistic action durations will not cause any constraint violations.

This paper presents significant improvements to the robust execution of PSTNs. Our approach is based on a recent, faster algorithm for finding negative cycles in non-DC STNUs. We also formally prove that many of the constraints included in others' work are unnecessary and that our algorithm can take advantage of a flexible real-time execution algorithm to react to observations of contingent durations that may fall outside the fixed STNU bounds. The paper presents an empirical evaluation of our approach that provides evidence of its effectiveness in robustly executing PSTNs derived from a publicly available benchmark.

2012 ACM Subject Classification Computing methodologies → Temporal reasoning; Theory of computation → Dynamic graph algorithms

Keywords and phrases Temporal constraint networks, probabilistic durations, dispatchable networks

Digital Object Identifier 10.4230/LIPIcs.TIME.2024.9

Supplementary Material *Software (Source code)*: `https://profs.scienze.univr.it/~posenato/software/cstnu/` [26]

1 Introduction

In many sectors of real-world industry, it is necessary to plan and schedule tasks allocated to agents participating in complex processes [19, 1]. Temporal planning aims to schedule tasks while respecting temporal constraints such as release times, maximum durations, and deadlines, which requires quantitative temporal reasoning. Over the years, major application developers have highlighted the need for explicit representation of actions with uncertain durations; and efficient algorithms for checking whether plans involving such actions are controllable, and for converting such plans into forms that enable them to be executed in real time with minimal computation, while preserving maximum flexibility.

A Probabilistic Simple Temporal Network (PSTN) is a formalism for representing and reasoning about actions subject to temporal constraints, where some action durations may be uncontrollable, modeled using continuous probability density functions. Recent work aims to manage this kind of uncertainty during execution by:

1. computing a dynamically controllable (DC) Simple Temporal Network with Uncertainty (STNU) whose bounded action durations capture as much of the combined probability mass of the corresponding probabilistic durations as possible;
2. deriving a dynamic execution strategy for the approximating STNU; and
3. using that strategy to execute the PSTN, hoping that its probabilistic action durations will not cause any constraint violations.

31st International Symposium on Temporal Representation and Reasoning (TIME 2024).
Editors: Pietro Sala, Michael Sioutis, and Fusheng Wang; Article No. 9; pp. 9:1–9:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Since unlikely action durations may nonetheless occur, this approach incurs a non-zero risk of failure. The typical goal is to minimize this risk, although some have sought to optimize a different objective function while accepting a pre-determined bound on the risk of failure.

This paper presents significant improvements to this approach that derive from recent, faster algorithms for solving several closely related problems, as well as some new theoretical results:

1. Since the iterative process of computing a DC STNU to approximate a PSTN relies on efficiently finding negative cycles in non-DC STNUs so that they can be resolved (e.g., by tightening the bounds on participating contingent durations), this paper uses a recent, faster algorithm for finding such cycles (Algorithm `FindSRNC` [16]). Its compact representation of such cycles avoids exponential blow-up. Like some recent work, our approximating algorithm (Algorithm `genApproxSTNU`) uses a general-purpose non-linear optimization solver to aid in this process; however, `genApproxSTNU` explicitly aims to maximize the combined probability mass of the probabilistic durations captured by the STNU's contingent durations. We also formally prove that many constraints included in others' work are unnecessary.
2. Given an approximating DC STNU, we then propose to use a recent, fast algorithm (Algorithm $\texttt{minDisp}_{\text{ESTNU}}$ [17]) to compute an equivalent dispatchable STNU having a minimal number of edges. Doing so allows the use of a flexible and efficient real-time execution strategy, implemented by the algorithm RTE* [18], instead of, for example, the inflexible earliest-first strategy used by many researchers.
3. Hence, we propose to execute the PSTN using RTE* to exploit the strategy's flexibility to react to observations of contingent durations that may fall outside the fixed STNU bounds.

The paper presents an empirical evaluation of our approach that provides evidence of its effectiveness in robustly executing PSTNs derived from a publicly available benchmark. In particular, it shows that taking advantage of a flexible real-time execution algorithm can increase the chances of successful executions.

2 Background

In this section, we recall the basic concepts and results about Simple Temporal Networks, Simple Temporal Networks with Uncertainty (STNUs), Probabilistic Simple Temporal Networks (PSTNs) and the known methods for approximating PSTNs by STNUs.

2.1 Simple Temporal Networks

A *Simple Temporal Network* (STN) is a pair $(\mathcal{T}, \mathcal{C})$ where $\mathcal{T}$ is a set of real-valued variables called timepoints; and $\mathcal{C}$ is a set of *ordinary* constraints, each of the form $(Y - X \leq \delta)$ for $X, Y \in \mathcal{T}$ and $\delta \in \mathbb{R}$ [5]. An STN is *consistent* if it has a solution as a constraint satisfaction problem (CSP). Each STN has a corresponding graph where the timepoints serve as nodes, and the constraints correspond to labeled, directed edges. In particular, each constraint $(Y - X \leq \delta)$ corresponds to an edge $X \xrightarrow{\delta} Y$ in the graph. Such edges may be notated as (X, δ, Y) for convenience.

A flexible and efficient *real-time execution* (RTE) algorithm has been defined for STNs that maintains time windows for each timepoint and, as each timepoint X is executed, only propagates constraints *locally,* to *neighbors* of X in the STN graph [28, 24]. An STN is called *dispatchable* if that RTE algorithm is guaranteed to satisfy all of the STN's constraints

Figure 1 A semi-reducible path (shaded gray on the left) and a Semi-Reducible Negative (SRN) cycle (shaded gray on the right).

no matter which execution decisions are made subject to the time-window constraints. Algorithms for generating equivalent dispatchable STNs have been presented [28, 24].

2.2 Simple Temporal Networks with Uncertainty

A *Simple Temporal Network with Uncertainty* (STNU) augments an STN to include *contingent links* that represent actions with uncertain, but bounded durations [23]. An STNU is a triple $(\mathcal{T}, \mathcal{C}, \mathcal{L})$ where $(\mathcal{T}, \mathcal{C})$ is an STN, and $\mathcal{L}$ is a set of contingent links, each of the form (A, x, y, C), where $A, C \in \mathcal{T}$ and $0 < x < y < \infty$. The semantics of STNU execution ensure that regardless of when the *activation timepoint* A is executed, the *contingent timepoint* C will occur such that $C - A \in [x, y]$. Thus, the duration $C - A$ is uncontrollable but bounded. The graph of an STNU $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L})$ is the graph of the STN $(\mathcal{T}, \mathcal{C})$ augmented to include *labeled* edges representing the contingent durations. In particular, each contingent link (A, x, y, C) has two corresponding edges in the STNU graph: a *lower-case* (LC) edge $A \xrightarrow{c:x} C$, notated as $(A, c{:}x, C)$, representing the uncontrollable possibility that the duration might take on its minimum value x; and an *upper-case* (UC) edge $C \xrightarrow{C:-y} A$, notated as $(C, C{:}{-y}, A)$, representing the possibility that it might take on its maximum value y.

The most important property of an STNU is whether it is *dynamically controllable* (DC). An STNU is *dynamically controllable* (DC) if there exists a dynamic, real-time execution strategy that guarantees that all constraints in $\mathcal{C}$ will be satisfied no matter how the contingent durations turn out [23, 10]. A strategy is dynamic because its execution decisions can react to observations of contingent executions without advance knowledge of future events. Morris [21] proved that an STNU is DC if and only if it does not include any *semi-reducible* negative cycles (SRN cycles). A path $\mathcal{P}$ is semi-reducible if certain constraint-propagation rules can be used to provide new edges that effectively *bypass* each occurrence of an LC edge in $\mathcal{P}$. As an example of a semi-reducible path and an SRN cycle, consider Figure 1. In the left network, the path $\Pi = (A, c{:}1, C, -1, B)$ is semi-reducible because it is possible to combine constraints $(A, c{:}1, C)$ and $(C, -1, B)$ to create an equivalent constraint $(A, 0, B)$ (dashed red) that bypasses $(A, c{:}1, C)$ in Π. In the right network, the path (cycle) $\Pi = (A, c{:}1, C, -1, D, D{:}{-10}, B, 7, A)$ is an SRN cycle because as before, it is possible to bypass $(A, c{:}1, C)$ by constraint $(A, 0, D)$ (dashed red), and the value of the resulting cycle $(A, 0, D, D{:}{-10}, B, 7, A)$ (sum of constraint values discarding possible labels) is negative. Indeed, this network is not DC because A must be executed after or as soon as D occurs to satisfy $(A, 0, D)$, and in the case that the contingent link $(B, 1, -10, D)$ duration outcomes to be 10, the constraint $(B, 7, A)$ will be violated.

In 2014, Morris [22] presented the first $O(n^3)$-time DC-checking algorithm.[1] In 2018,

[1] As is common in the literature, we use n for the number of timepoints, m for the number of ordinary constraints; and k for the number of contingent links.

Cairo *et al.* [2] presented their $O(mn + k^2n + kn \log n)$-time `RUL⁻` algorithm. In 2022, Hunsberger and Posenato [14] subsequently presented a faster version, called `RUL2021`, that has the same worst-case complexity but achieves an order-of-magnitude speedup in practice by restricting the edges it inserts into the network during constraint propagation.

Following the literature, we refer to ordinary or LC edges as LO-edges and ordinary or UC edges as OU-edges. An ESTNU graph has the form $(\mathcal{T}, \mathcal{E}_{\rm o} \cup \mathcal{E}_{\rm lc} \cup \mathcal{E}_{\rm uc} \cup \mathcal{E}_{\rm ucg})$, where $\mathcal{E}_{\rm o}$ is the set of ordinary edges, $\mathcal{E}_{\rm lc}$ and $\mathcal{E}_{\rm uc}$ are the sets of LC and UC edges, and $\mathcal{E}_{\rm ucg}$ is the set of generated *wait* edges (described later). The graphs, $\mathcal{G}_{\ell o}$ and $\mathcal{G}_{ou}$, of the LO- and OU-edges, respectively, can be viewed as STNs by ignoring the alphabetic labels on LC or UC edges.

2.3 Probabilistic Simple Temporal Networks

A *Probabilistic Simple Temporal Network* (PSTN) is similar to an STNU, except that each contingent duration, $C - A$, is modeled as a random variable with a specified probability density function (pdf) p [27, 7]. This paper assumes that each probabilistic duration has a log-normal distribution.[2]

Since pdfs can have infinite tails, successfully executing a PSTN cannot be guaranteed in general. Instead, researchers have focused on approximating PSTNs by STNUs [7, 33, 30, 31]. *The approximating STNU differs from the PSTN only in representing the contingent durations; the ordinary constraints all stay the same.* The aim is to choose bounds for the approximating STNU's contingent links that capture as much probability mass of the probabilistic durations as possible while preserving the STNU's controllability. For example, if (A, x, y, C) is a contingent link approximating a probabilistic duration (A, C, p), then the probability mass *captured* by the contingent link is $\int_x^y p(t)dt = F(y) - F(x)$, where F is the associated cumulative distribution function (cdf).

2.3.1 Approximating PSTNs by Strongly Controllable STNUs

Early work sought to approximate PSTNs by *strongly controllable* STNUs. (An STNU $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L})$ is *strongly controllable* (SC) if there exists a fixed schedule for its controllable timepoints that guarantees that all constraints in $\mathcal{C}$ will be satisfied no matter how the durations of the contingent links in $\mathcal{L}$ turn out.) Tsamardinos [27] aimed to find a fixed schedule for a PSTN that maximized the probability that all of its constraints would be satisfied. However, his approach was too restrictive: it did not allow ordinary constraints between pairs of contingent timepoints.

Fang *et al.* [7] defined a similar problem, called the *chance-constrained probabilistic Simple Temporal Problem* (cc-pSTP). Instead of aiming to minimize the risk of failure, the cc-pSTP is the problem of finding a static schedule that optimizes a given objective function (e.g., complete all tasks as early as possible) while keeping the risk of failure below a given bound (e.g., less than 5 percent). In other words, the cc-pSTP accepts a bounded risk of failure (a.k.a. a *chance constraint*). To solve the cc-pSTP, they create an initial approximating STNU in which the bounds on each contingent link are *variables,* not constants. Their algorithm then applies constraint-propagation/edge-generation rules (a.k.a. reduction rules) to enforce the SC property. These rules are generalized from prior work on strong controllability [29, 27] to accommodate the bounds on the contingent links being variables instead of constants.

---

[2] Chen *et al.* [3] observed that "Existing experiments data ... showed that heavy-tailed distributions, such as lognormal, best fit the task uncertainty introduced by humans in collaborative tasks [6]. This is corroborated by work that showed the human reaction time is also best modeled as log-normal [32]."

The result is at most n^2 linear constraints, each involving the contingent link bounds-as-variables. In contrast, the chance constraint is non-linear since it depends on the cdfs for the probabilistic durations. They approximate the chance constraint using Boole's inequality, which does not require assuming independence of the probabilistic durations, as follows: (actual probability of failure) $\leq \sum_{i=1}^{k}(F_i(x_i)+(1-F_i(y_i)) \leq \Delta$, where each F_i is a the cdf for the i^{th} probabilistic link, and Δ is the given bound on the risk of failure. The objective function, which is provided as an input, can also be non-linear. After constructing their non-linear optimization problem, they solve it using an off-the-shelf solver, called SNOPT [9].

Wang and Williams [30] presented the *Rubato* algorithm, which tackles the cc-pSTP by *decoupling* the risk-allocation problem (i.e., assigning fixed bounds to the STNU's contingent links) from strong-controllability checking. In this way, the risk-allocation problem, solved by a non-linear solver, need not include the $O(n^2)$ constraints generated by the previously mentioned constraint-propagation rules, keeping the optimization problem small. Once risk allocation is done, the SC checker is run which, in negative instances, outputs a simple negative cycle. In such cases, they then accumulate a new constraint stipulating that that cycle must be made non-negative. They iteratively run this risk-allocation/SC-checking process until an SC STNU is found, which then yields a static schedule for the PSTN.

2.3.2 Approximating a PSTN by a Dynamically Controllable STNU

Wang [31] defined a dynamic version of the cc-pSTP that aims to approximate a PSTN by a DC STNU. Analogous to *Rubato*, Wang used an iterative approach that decouples risk-allocation from DC checking. For the first risk-allocation step, a non-linear optimization solver generates initial bounds for the STNU's contingent durations that capture as much of the probability mass of the PSTN's probabilistic durations as possible while also satisfying the ordinary constraints from the STNU. For the DC-checking step, Morris' $O(n^4)$-time DC-checking algorithm is modified so that it outputs an SRN cycle for non-DC networks. Wang noted that such cycles may not be simple, but presented no details on how to compute or represent them. (In the worst case, SRN cycles can involve exponentially many edges [12].)[3]

If the candidate STNU happens to be non-DC, it must contain an SRN cycle, which can be resolved by making it non-negative or non-semi-reducible. Following Morris [21], Wang noted that semi-reducibility requires that each LC edge can be *reduced away* by a (negative-length) *extension subpath*.[4] Thus, he argued that modifying any one of the participating extension sub-paths by making it non-negative would cause the entire cycle to be non-semi-reducible. (However, as shown below, this is often not the case.) Thus, Wang's approach to resolving an SRN cycle involved accumulating a *disjunction* of potentially very many new constraints, one for each participating extension subpath. Hence, his approach requires the use of a disjunctive linear program solver. Although he gives some empirical evaluations, only very high-level implementation details are provided, making the results difficult to evaluate.

3 Yu, Fang and Williams [33] addressed resolving a non-DC STNU by finding an SRN cycle within it and then tightening the bounds on participating contingent durations. However, unlike Wang, they failed to recognize that individual labeled edges can appear multiple times in an SRN cycle.

4 An *extension subpath* for an LC edge e in a path $\mathcal{P}$ is a negative-length subpath $\mathcal{P}_e$ that immediately follows e in $\mathcal{P}$ and such that the constraint-propagation/edge-generation rules given by Morris [21] can be used to generate a new edge E that effectively bypasses e in $\mathcal{P}$.

3 Preliminary Steps

In this section, we introduce some preliminary results that allow the determination of a new algorithm for a robust execution of PSTNs.

3.1 Efficiently Finding and Representing SRN Cycles

Iteratively finding a DC STNU to approximate a PSTN typically requires numerous calls to an algorithm for finding SRN cycles in non-DC STNUs. For this, Wang used a modified version of Morris' $O(n^4)$-time DC-checking algorithm. Instead, this paper takes advantage of a new, faster $O(mn + kn^2 + kn \log n)$-time algorithm, `FindSRNC`, for finding *and compactly representing* SRN cycles [16]. Aside from its greater speed, there are two main features that are important for this paper. First, because an *indivisible* SRN cycle in a non-DC STNU can have, in the worst case, an *exponential number of occurrences* of LC and UC edges [12], the output of `FindSRNC` includes a hash table that compactly represents the repeating structures that necessarily occur in such cycles, while requiring only $O(mk + k^2n)$ space. Second, `FindSRNC`, like the `RUL2021` algorithm [14] on which it is based, detects three different kinds of SRN cycles: (1) a negative cycle in the LO-graph; (2) a special kind of cycle, called a *CC loop*; and (3) a cycle arising from a cycle of interruptions of its recursive processing of UC edges. The following section recalls how Wang's approach to resolving SRN cycles introduces potentially very many disjunctive constraints and then rigorously addresses the different ways that each kind of SRN cycle returned by `FindSRNC` can be resolved, in one case without requiring any disjunctions, in another case requiring only a single disjunction, and in a third case requiring a bounded number of disjunctions.

3.2 More Efficient Resolution of SRN Cycles

To resolve an SRN cycle L, Wang generates a *disjunctive* collection of linear constraints. The main constraint is to make $|L|$ non-negative. The other constraints, which can be numerous, aim to make L non-semi-reducible by, for each *occurrence* of an LC edge e in L, constraining its *extension subpath* $\mathcal{P}_e$ to be non-negative. (Each occurrence of an LC edge in L can have a very different extension subpath.) The idea is that if *any* of these constraints are satisfied, then L will either be non-negative or non-semi-reducible (or both). However, while it is true that modifying an extension subpath $\mathcal{P}_e$ by making it non-negative renders it unable to reduce away the LC edge, it does not necessarily make L non-semi-reducible. Why? Because other edges following $\mathcal{P}_e$ in L might combine with $\mathcal{P}_e$ to create a new extension subpath for e, as illustrated below.

$$\cdots \quad A \xrightarrow{c:5} C \xrightarrow{1} A' \xrightarrow{c':4} C' \xrightarrow{-6} F \xrightarrow{-2} G \xrightarrow{-3} H \quad \cdots$$

In this example, the extension subpath for the LC edge $e = (A, c{:}5, C)$ is the negative-length subpath from C to F, shaded dark gray. This subpath can be made non-negative by increasing the lower bound on the LC edge $(A', c'{:}4, C')$ from 4 to 5. However, doing so would not make the overall path non-semi-reducible because the path from C to G, shaded light gray, would still be negative and hence could be used to reduce away e. As a result, a subsequent iteration of Wang's algorithm might return the very same SRN cycle, albeit with a slightly different length. Even worse, a chain of negative edges following an existing extension subpath for e might lead to numerous nearly identical iterations. Furthermore, a single SRN cycle might have many LC edges leading to numerous disjunctive constraints, thereby compounding the problem for the disjunctive optimization solver, making it expensive for larger networks.

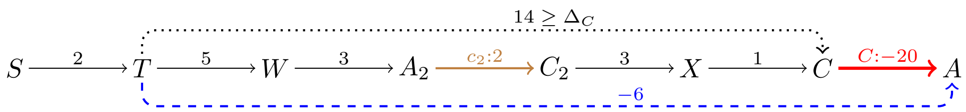

Figure 2 Generating a (blue, dashed) bypass edge for a (red) UC-edge, assuming that $\Delta_C = 12$

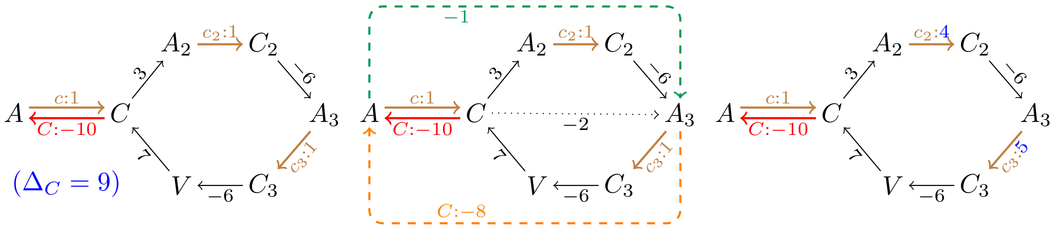

Figure 3 A CC loop (left); a CC-based SRN cycle (center); and resolving the SRN cycle (right)

3.3 Three Kinds of SRN Cycles Computed by `FindSRNC`

Before addressing how to resolve the SRN cycles output by `FindSRNC`, we must discuss how `FindSRNC` works. As shown in Figure 2, `FindSRNC` processes each UC edge $\mathbf{E} = (C, C{:}{-y}, A)$, propagating backward from C along LO-edges aiming to generate edges that effectively *bypass* $\mathbf{E}$. Back-propagation continues while the subpath being explored has length less than $\Delta_C = y - x$. If that distance ever becomes greater than or equal to Δ_C, as in the path from T to C in Figure 2, then a bypass edge, shown as blue and dashed, is generated, and back-propagation stops.

As in Johnson's algorithm [4], the back-propagation is guided by a potential function that is a solution to the graph of LO-edges viewed as an STN. The potential function is initialized by a call to Bellman-Ford [4] and, after the processing of each UC edge, is updated to accommodate any newly generated edges. If the updating reveals a negative cycle in the LO-graph, then the STNU cannot be DC. Therefore, `FindSRNC` outputs that negative cycle.

There are two ways that `FindSRNC`'s back-propagation can be blocked: (1) by a *CC loop*, or (2) by bumping into another UC edge. A CC loop is where back-propagation from C cycles back to C with all encountered distances less than Δ_C, as illustrated on the lefthand side of Figure 3. A CC loop does not necessarily entail an SRN cycle, but it can: if there exists a negative-length LO-path emanating from C that can be used to reduce away the LC edge $(A, c{:}x, C)$ [14]. An example of this is shown in the center of Figure 3. Based on the edge-generation rules from Morris [21], the negative-length (dotted) path from C to A_3 can be used to generate the (dashed, green) bypass edge $(A, -1, A_3)$. Meanwhile, the path from A_3 to A can be used to generate the (dashed, orange) *wait* edge $(A_3, C{:}{-8}, A)$, thereby forming a negative cycle in the OU-graph, which implies that the network cannot be DC. In such a case, `FindSRNC` outputs the SRN cycle formed by the matching LC and UC edges together with the CC loop. We call such a cycle a *CC-based* SRN cycle for convenience.

Back-propagation from C can also be blocked by bumping into another UC edge, say $\mathbf{E}_2$, while encountered distances remain less than Δ_C. In such cases, $\mathbf{E}$'s processing is interrupted until $\mathbf{E}_2$ is fully processed. Once all edges bypassing $\mathbf{E}_2$ have been generated, back-propagation from C continues. But if a *cycle* of such interruptions is found, all processing is blocked, and the network cannot be DC [2]. In that case, `FindSRNC` returns the SRN cycle formed by concatenating the interrupted subpaths, including the corresponding UC edges, as shown on the left of Figure 4, where it is assumed that the length of each LC edge is 1.

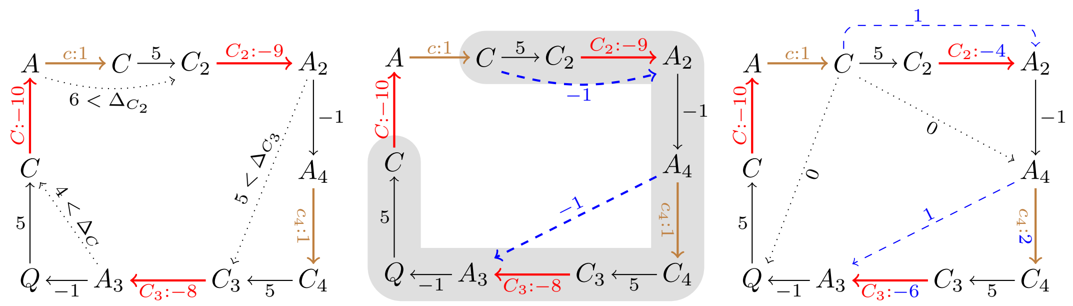

Figure 4 A cycle of interruptions (left); a weakened version with a (shaded) CC loop (center); making it non-semi-reducible by constraining subpaths emanating from C to be non-negative (right)

3.4 Resolving SRN Cycles Output by `FindSRNC`

SRN cycles are, by definition, *negative* and *semi-reducible,* so such cycles can be resolved by making them non-negative or non-semi-reducible. As in earlier work, we restrict attention to resolving an SRN cycle by increasing the lengths of LC or UC edges contained within it (i.e., by tightening the bounds on the corresponding contingent links). Although the bypass edges computed by `FindSRNC` are invariably ordinary, the paths they bypass may have multiple LC and UC edges. Increasing the lengths of those LC or UC edges in turn increases the lengths of the bypass edges.

Since resolving an SRN cycle by making it non-negative is always an option, this section focuses on cases where an SRN cycle can be made non-semi-reducible without making it non-negative. The lemmas below address the three kinds of SRN cycles output by `FindSRNC`.

▶ **Lemma 1.** *If an SRN cycle comprises only LO-edges, then the only way to resolve the cycle is by making it non-negative.*

Proof. A negative cycle comprising only LO-edges is necessarily semi-reducible [13]. ◀

▶ **Lemma 2.** *Let L be a CC-based SRN cycle where $(C, C{:}{-y}, A)$ and $(A, c{:}x, C)$ are the relevant UC and LC edges. Then, the only way to make L non-semi-reducible is by making the length of each subpath emanating from C in the CC loop non-negative.*

The righthand side of Figure 3 shows an example of making a CC-based SRN cycle non-semi-reducible, in this case, by increasing the lengths of the LC edges A_2C_2 and A_3C_3 to ensure that every subpath emanating from C is non-negative. (The modified lengths are shown in blue.) Notice that the length of the entire CC-based cycle is still negative: -2.

Proof. If any subpath emanating from C in the CC loop has negative length, then it can be used to reduce away (bypass) the LC edge $(A, c{:}x, C)$, preserving the SRN cycle [14]. ◀

Although each subpath emanating from C needs to be non-negative, that need not require an explicit constraint for each timepoint following C. First, since the only allowed modifications involve lengthening edges, any subpath emanating from C that is already non-negative in L does not need to be explicitly constrained. In addition, if a subpath from C to X is constrained to be non-negative, and the path from X to Y is non-negative, then the subpath from C to Y will automatically be non-negative. A one-time traversal of the edges in L suffices to determine the conjunction of constraints needed to make L non-semi-reducible.

▶ **Lemma 3.** *Let L be an SRN cycle obtained from a cycle of interruptions of processings of UC edges (e.g., as shown on the lefthand side of Figure 4). If $\mathbf{E} = (C, C{:}{-y}, A)$ and $e = (A, c{:}x, C)$ are adjacent in L, then L can be made non-semi-reducible by making the length of each subpath emanating from C that does* not *include $\mathbf{E}$ non-negative. Although there can be multiple pairs of adjacent labeled edges providing such opportunities for making L non-semi-reducible, there are no other ways of making L non-semi-reducible.*

Proof. A cycle of interruptions necessarily entails an SRN cycle [2], so resolving L requires breaking that cycle of interruptions. One way is to lengthen edges in L enough to enable the generation of bypass edges for all UC edges in L. But that would yield a cycle comprising only LO-edges and, since negative LO-cycles are invariably semi-reducible, resolving the SRN cycle in this way would still require making $|L|$ non-negative. The only other outcome that can arise from increasing the lengths of edges preceding a UC edge would be the creation of a CC loop, as illustrated in Figure 4 (center), where the UC edges C_2A_2 and C_3A_3 have been bypassed by dashed, blue edges, creating a CC loop from C back to C. Since a CC loop contains only LO-edges, a CC loop can only be created if all other UC edges have been bypassed.

Claim: Constraining every subpath emanating from C that terminates at or before the UC edge $(C, C{:}{-y}, A)$, as illustrated on the righthand side of Figure 4, will ensure that L is non-semi-reducible. (In the figure, constraining the subpath from C to A_4 to be non-negative automatically ensures that the subpaths terminating at A_2, C_4 and C_3 will also be non-negative, given the negative edge from A_2 to A_4, and the non-negative paths from A_4 to C_4 and C_3. Similarly, constraining the subpath from C to Q to be non-negative ensures that the subpaths terminating at A_3 and C will also be non-negative.)

Proof of Claim. If every subpath emanating from C is non-negative, then every UC edge other than $(C, C{:}{-y}, A)$ must be bypassable. For example, the first encountered UC edge $(C', C'{:}{-y'}, A')$ must be bypassable since the subpath from C to A' being non-negative implies that the subpath from C to C' must be at least $y' > \Delta_{C'}$. An inductive argument ensures that all following UC edges are bypassable. But then Lemma 2 ensures that the CC-based cycle formed using those bypass edges is non-semi-reducible. (End proof of claim.)

Finally, if any subpath emanating from C is negative, then the LC edge $(A, c{:}x, C)$ can be bypassed, yielding a cycle of interruptions that cannot be resolved via a CC loop involving $(C, C{:}{-y}, A)$ and $(A, c{:}x, C)$; hence the only options for making L non-semi-reducible must involve forming a CC loop using a different pair of adjacent, matching UC and LC edges. ◀

Summary. All three types of SRN cycles L returned by `FindSRNC` can be resolved by making L non-negative. Alternatively, L can be made non-semi-reducible if: (1) it is a CC-based SRN cycle for a contingent timepoint C, where each subpath emanating from C is non-negative; or (2) L arises from a cycle of interruptions and L includes at least one adjacent pair of matching UC and LC edges. This analysis of SRN cycles greatly reduces the need for disjunctive constraints as compared to the approach of Wang. It also avoids the problem of repeatedly revisiting the same SRN cycle, when making the length of an extension subpath non-negative, fails to make it non-semi-reducible. Finally, we conjecture that occurrences of CC loops and (especially) cycles of interruptions that can be weakened to reveal a CC loop will occur only rarely in practice and, therefore, our new algorithm, presented in Section 4, focuses exclusively on constraining the SRN cycle itself to be non-negative (i.e., a single constraint).

4 New Algorithm for Robustly Executing PSTNs

Given any PSTN, our new algorithm for robustly executing PSTNs: (1) computes an approximating STNU that is DC, using the `FindSRNC` algorithm to efficiently compute and compactly represent SRN cycles in non-DC STNUs; (2) converts that STNU into an equivalent dispatchable ESTNU; and (3) executes the original PSTN using the RTE* algorithm, leveraging its flexibility to react to possibly extreme contingent durations.

Algorithm 1 `genApproxSTNU`: generate a DC STNU that approximates a given PSTN

Input: $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{M})$: a PSTN where $\mathcal{M} = \{(A_i, C_i, \text{Lognormal}(\mu_i, \sigma_i)) \mid i \in \{1, \ldots, k\}\}$
Output: $(\mathcal{S}_u, F)$, where $\mathcal{S}_u = (\mathcal{T}, \mathcal{C}, \mathcal{L})$ is an approximating DC STNU for $\mathcal{S}$, and F is the joint probability mass of the durations in $\mathcal{M}$ captured by the links in $\mathcal{L}$). Or $\perp$ if unable.

// Initialize the approximating STNU
1 $\mathcal{S}_u := (\mathcal{T}, \mathcal{C}, \mathcal{L})$, where $\mathcal{L} = \{(A_i, x_i = e^{\mu_i - 3.3\sigma_i}, y_i = e^{\mu_i + 3.3\sigma_i}, C_i) \mid i \in \{1, \ldots, k\}\}$
2 $(L, \mathcal{H}) :=$ `FindSRNC(copy(`$\mathcal{S}_u$`))` // L = SRN cycle; $\mathcal{H}$ = edge-annotation hash table
3 **while** L **do**
   // Below, `len` $= |L|$; a_i, b_i = num. occurrences of ith LC, UC edges in (fully expanded) L
4    $($`len`$, (a_1, \ldots, a_n), (b_1, \ldots, b_n)) :=$ `fetchEdgeInfo(negCycle, edgeAnnHash)`
5   **if** $\Sigma_{i=1}^{k}(a_i + b_i) == 0$ **then return** $\perp$ // No labeled edges in expanded SRN cycle
6    $\mathcal{A} := \{i \mid a_i > 0$ **or** $b_i > 0\}$ // Collect indices of contingent links participating in SRN cycle
7    $\kappa := |\mathcal{A}|$ // $\kappa \leq k$ is num. contingent links participating in SRN cycle
8   Let $\pi\colon \{1, \ldots, \kappa\} \mapsto \mathcal{A}$ be a re-ordering of the indices of $\mathcal{A}$ from 1 to κ
9   `bounds` $:= (x_{\pi(1)}, y_{\pi(1)}, \ldots, x_{\pi(\kappa)}, y_{\pi(\kappa)})$
10   `muVec` $:= (\mu_{\pi(1)}, \ldots, \mu_{\pi(\kappa)})$; `sigVec` $:= (\sigma_{\pi(1)}, \ldots, \sigma_{\pi(\kappa)})$
11   `coeffs` $:= (a_{\pi(1)}, -b_{\pi(1)}, \ldots, a_{\pi(\kappa)}, -b_{\pi(\kappa)})$
12   `const` $:= -$`len` $+ \sum_{1 \leq i \leq \kappa}(a_{\pi(i)} x_{\pi(i)} - b_{\pi(i)} y_{\pi(i)})$
13    $((\hat{x}_{\pi(1)}, \hat{y}_{\pi(1)}, \ldots, \hat{x}_{\pi(\kappa)}, \hat{y}_{\pi(\kappa)}), \hat{F}) :=$ `nlpOpt(`κ`, muVec, sigVec, coeffs, const, bounds)`
14   **if** $\hat{F} == \perp$ **then return** $\perp$
15   **foreach** $i \in \{1, \ldots, \kappa\}$ **do** $x_{\pi(i)} := \hat{x}_{\pi(i)}$ **and** $y_{\pi(i)} := \hat{y}_{\pi(i)}$ // Update bounds
16   `(negCycle, edgeAnnHash) := FindSRNC(copy(`$\mathcal{S}_u$`))` // Prepare for next iteration

// $\mathcal{S}_u$ is dynamically controllable. Re-compute objective function over all contingent links.
17 $F := \Pi_{1 \leq i \leq k}\,(\texttt{lnCDF}(y_i, \mu_i, \sigma_i) - \texttt{lnCDF}(x_i, \mu_i, \sigma_i))$
18 **return** $(\mathcal{S}_u, F)$, *where* $\mathcal{S}_u$ *has updated bounds* $(x_1, y_1, \ldots, x_k, y_k)$

4.1 Generating a DC STNU to Approximate a PSTN

The `genApproxSTNU` algorithm (Algorithm 1) takes as its input a PSTN $\mathcal{S}$ with k probabilistic durations of the form $(A_i, C_i, \text{Lognormal}(\mu_i, \sigma_i))$.[5] It aims to generate an approximating STNU for $\mathcal{S}$ that is DC by providing bounds for the contingent links that maximize the joint probability mass of the probabilistic durations they capture while preserving the DC property.

At Line 1, the approximating STNU is initialized by setting the bounds for each contingent link (A_i, x_i, y_i, C_i) to $x_i = e^{\mu_i - 3.3\sigma_i}$ and $y_i = e^{\mu_i + 3.3\sigma_i}$, which represent ± 3.3 standard deviations for the underlying normal distribution, which ensures capturing approximately 99.96% of the probability mass. As a result, we expect that the initial STNU will *not* be DC.

Next, at Line 2, it calls the `FindSRNC` algorithm on a *copy* of the STNU. (`FindSRNC` destructively modifies its input.) For non-DC STNUs, `FindSRNC` outputs a compact repre-

[5] In other words, $\text{Lognormal}(\mu_i, \sigma_i) = e^{\mu_i + \sigma_i Z}$, where Z is a standard normal random variable.

sentation of an SRN cycle as a pair, $(L, \mathcal{H})$, where L is a list of ordinary, LC and UC edges with no repeats, and $\mathcal{H}$ is an *edge-annotation* hash table [16]. Although L has no repeat edges, some of its ordinary edges may be *bypass* edges. Each bypass edge E in L has an entry in the hash table $\mathcal{H}$ that identifies the path $\mathcal{P}_E$ bypassed by E. In addition, the bypassed paths may recursively include other bypass edges. In the worst case, fully expanding L by recursively replacing each occurrence of a bypass edge by the path it bypassed can lead to an exponential number of edges due to the presence of repeated structures [11]. In contrast, the edge-annotation hash table uses only $O(k^2n)$ space to store the relevant information [16].

As long as `FindSRNC` returns an SRN cycle, the `while` loop at Lines 3–16 aims to resolve the cycle by tightening the bounds on the participating contingent links while retaining as much of the probability mass from the corresponding probabilistic durations as possible. Each iteration begins, at Line 4, by calling the `fetchEdgeInfo` algorithm (Algorithm 2) which returns the following information: `len`, the length of (one traversal of) the SRN cycle; and two vectors $(a_1, \ldots, a_k)$ and $(b_1, \ldots, b_k)$, where each a_i specifies the number of occurrences of the LC edge $(A_i, c_i{:}x_i, C_i)$ in the (fully expanded) SRN cycle, and each b_i the number of occurrences of the UC edge $(C_i, C_i{:}{-y_i}, A_i)$. Crucially, as will be seen later, this can be done in $O(nk^2)$ time, even if the (fully expanded) cycle contains an exponential number of edges.

At Line 5, if there are no labeled edges in the (fully expanded version of) the SRN cycle, `genApproxSTNU` returns $\perp$, since such a cycle cannot be resolved by adjusting the bounds on contingent links. Otherwise, at Lines 6–12, it prepares data for the the constraint optimization problem of finding new bounds for the contingent links that maximize the captured joint probability mass subject to the constraint of making the SRN cycle non-negative.

At Line 6, the set $\mathcal{A}$ collects the indices i for the contingent links whose labeled edges participate in the SRN cycle L. At Line 7, $\kappa = |\mathcal{A}| \leq k$ denotes the number of contingent links participating in L. Since resolving the SRN cycle only requires dealing with those κ contingent links, Line 8 specifies a bijection π from $\{1, 2, \ldots, \kappa\}$ to $\mathcal{A}$ that facilitates preparing data for the non-linear solver, focusing only on the participating contingent links.

Lines 9–10 collect, for each participating contingent link, the current values of the bounds, $x_{\pi(i)}$ and $y_{\pi(i)}$, and the μ_i and σ_i values of the associated log-normal distributions. Lines 11–12 collect information needed to specify the constraint, $|L| \geq 0$. First, `coeffs` collects the number of occurrences of the labeled edges from participating contingent links. These counts are important because, for example, increasing the value of some x_i to $\hat{x}_i$ increases $|L|$ by $a_i(\hat{x}_i - x_i)$, while decreasing y_i to $\hat{y}_i$ increases $|L|$ by $b_i(y_i - \hat{y}_i)$. Overall, changing the values in $(x_{\pi(1)}, y_{\pi(1)}, \ldots, x_{\pi(\kappa)}, y_{\pi(\kappa)})$ to $(\hat{x}_{\pi(1)}, \hat{y}_{\pi(1)}, \ldots, \hat{x}_{\pi(\kappa)}, \hat{y}_{\pi(\kappa)})$ increases $|L|$ by:

$$\textstyle\sum_{i=1}^{\kappa}(a_{\pi(i)}(\hat{x}_{\pi(i)} - x_{\pi(i)}) + b_{\pi(i)}(y_{\pi_(i)} - \hat{y}_{\pi(i)}))$$

Therefore, satisfying $|L| \geq 0$ requires choosing values, $\hat{x}_{\pi(i)}$ and $\hat{y}_{\pi(i)}$, such that:

$$\textstyle\sum_{i=1}^{\kappa}(a_{\pi(i)}\hat{x}_{\pi(i)} - b_{\pi(i)}\hat{y}_{\pi_(i)}) \;\geq\; -|L| + \sum_{i=1}^{\kappa}(a_{\pi(i)}x_{\pi(i)} - b_{\pi(i)}y_{\pi(i)})$$

The lefthand sum is a linear combination of the *variables,* $\hat{x}_{\pi(i)}$ and $\hat{y}_{\pi(i)}$, while the quantity on the righthand side is a constant. That constant is assigned to `const` at Line 12.

Line 13 calls a non-linear optimization solver, here called `nlpOpt`. Currently, our algorithm uses the `fmincon` solver provided by Matlab; others have used the SNOPT solver. If the solver is unable to find a new set of bounds for the contingent links to resolve the SRN cycle, then the entire algorithm fails. However, if successful, it returns a vector of the new bounds, $\hat{x}_i$ and $\hat{y}_i$, and the value of the objective function F. Line 15 updates the bounds in the STNU to reflect the new values. Line 16 calls `FindSRNC` in preparation for the next iteration of the `while` loop. If Line 18 is reached, then the STNU $\mathcal{S}_u$ has been made DC. It is returned by the algorithm, along with the updated value of the objective function.

Algorithm 2 `fetchEdgeInfo`

Input: k, the number of contingent links; $\mathcal{P}$, a path in an STNU graph; `edgeAnnHash`, a hash-table of $(E, \mathcal{P}_E)$ pairs where $\mathcal{P}_E$ is the path bypassed by the edge E

Output: $(\texttt{len}, (a_1, \ldots, a_k), (b_1, \ldots, b_k))$, where $\texttt{len} = |\mathcal{P}|$, and a_i and b_i are the numbers of times $(A_i, c_i{:}x_i, C_i)$ and $(C_i, C_i{:}{-y_i}, A_i)$ appear in the *fully unwound* version of $\mathcal{P}$

1 $\texttt{infoHash} :=$ new hash table; $\texttt{len} := 0$
2 $\texttt{lcCounts} := (0, \ldots, 0)$; $\texttt{ucCounts} := (0, \ldots, 0)$ // Counts of occurrences of LC/UC edges
3 **foreach** $E \in \mathcal{P}$ **do**
4   **if** $E = (A_i, c_i{:}x_i, C_i)$ *is an LC edge for some* i **then**
5     $\texttt{len} := \texttt{len} + x_i$; $\texttt{lcCounts}[i] := \texttt{lcCounts}[i] + 1$
6   **else if** $E = (C_i, C_i{:}{-y_i}, A_i)$ *is a UC edge for some* i **then**
7     $\texttt{len} := \texttt{len} - y_i$; $\texttt{ucCounts}[i] := \texttt{ucCounts}[i] + 1$
8   **else if** $\exists(E, \mathcal{P}_E) \in$ *edgeAnnHash* **then** // E is a bypass edge for path $\mathcal{P}_E$
9   **if** $\exists(E, \cdot) \in$ *infoHash* **then** // E has already been processed by `fetchEdgeInfo`
10       $(\texttt{len}', \texttt{lcCounts}', \texttt{ucCounts}') := \texttt{infoHash.getValue}(E)$
11   **else**
12       $(\texttt{len}', \texttt{lcCounts}', \texttt{ucCounts}') := \texttt{fetchEdgeInfo}(k, \mathcal{P}_E)$ // Recursively process $\mathcal{P}_E$
13       $\texttt{infoHash.setValue}(E, (\texttt{len}', \texttt{lcCounts}', \texttt{ucCounts}'))$ // Store results in `infoHash`
14     $\texttt{len} := \texttt{len} + \texttt{len}'$
15   **foreach** $i \in \{1, 2, \ldots, k\}$ **do**
16       $\texttt{lcCounts}[i] := \texttt{lcCounts}[i] + \texttt{lcCounts}'[i]$; $\texttt{ucCounts}[i] := \texttt{ucCounts}[i] + \texttt{ucCounts}'[i]$
17   **else** $\texttt{len} = \texttt{len} + |E|$ // E is an ordinary edge from the original STNU
18 **return** $(\texttt{len}, \texttt{lcCounts}, \texttt{ucCounts})$

The `fetchEdgeInfo` Algorithm

The `fetchEdgeInfo` algorithm (Algorithm 2) accumulates the numbers of occurrences of LC and UC edges in the SRN cycle L. Crucially, it does not need to expand L fully. Instead, it uses a hash table, `infoHash`, to keep track of the numbers of occurrences of labeled edges recursively hiding within each encountered bypass edge. When it first sees a bypass edge E, it recursively processes it, then stores the vectors of counts in the `infoHash` hash table. Subsequent encounters with E only need to do a constant-time look-up in the hash table (cf. Lines 9–13 in Algorithm 2). `fetchEdgeInfo` requires $O(nk^2)$ space due to at most $O(kn)$ entries stored in the `infoHash` hash table, each of size $O(k)$. This is less than the $O(n^2k)$ size of the edge-annotation hash table, $\mathcal{H}$, passed in as an input.

4.2 Flexible and Efficient Real-time Execution

Most DC-checking algorithms generate conditional *wait* constraints that must be satisfied by any valid execution strategy. Each wait is represented by a labeled edge of the form $(W, C{:}{-w}, A)$, which can be glossed as: "While C remains unexecuted, W must wait at least w after A." (Despite the similar notation, a wait is distinguishable from the original UC edge since its source timepoint is *not* the contingent timepoint C.) Morris [22] defined an *Extended STNU* (ESTNU) to be an STNU augmented with such waits. He then extended the notion of dispatchability to ESTNUs, defining an ESTNU to be dispatchable if all of its

STN projections are STN-dispatchable.[6] He then argued that a dispatchable ESTNU would necessarily provide a guarantee of flexible and efficient real-time execution.

Hunsberger and Posenato [18] later:

1. formally defined a flexible and efficient real-time execution algorithm for ESTNUs, called RTE*;
2. defined an ESTNU to be dispatchable if every run of RTE* necessarily satisfies all of the ESTNU's constraints; and
3. proved that an ESTNU satisfying their definition of dispatchability necessarily satisfies Morris' definition (i.e., all of its STN projections are STN-dispatchable).

The RTE* algorithm provides maximum flexibility during execution, unlike the *earliest-first* strategy used for non-dispatchable networks.

Most DC-checking algorithms do not generate dispatchable ESTNUs. However, Morris [22] argued that his $O(n^3)$-time DC-checking algorithm could be modified, without impacting its complexity, to generate a dispatchable output. In 2023, Hunsberger and Posenato [15] presented a faster, $O(mn + kn^2 + n^2 \log n)$-time ESTNU-dispatchability algorithm. However, neither of these algorithms provides any guarantees about the number of edges in the dispatchable output. More recently, Hunsberger and Posenato [17] presented $\texttt{minDisp}_{\text{ESTNU}}$, the first ESTNU-dispatchability algorithm that, in $O(kn^3)$ time, generates an equivalent dispatchable ESTNU *having a minimal number of edges,* which is important since it directly affects the real-time computations of the RTE* algorithm.

Our new approach to executing PSTNs in real time is the first to explore the use of the flexible and efficient RTE* algorithm. To enable this, we first use the $\texttt{minDisp}_{\text{ESTNU}}$ algorithm to convert the DC STNU output by `genApproxSTNU` into an equivalent, dispatchable ESTNU having a minimal number of edges. Then, we execute the PSTN using the RTE* algorithm as if it were being applied to the dispatchable ESTNU. In other words, the time-windows and wait constraints maintained by RTE* are determined by the ESTNU's edges. In addition, to increase the chances of successful execution, RTE* is run not with the needlessly inflexible *earliest-first* strategy that has been used by others [3, 31, 8], but with a more flexible *midpoint* strategy made available by RTE*. In particular, if a currently enabled timepoint X has a time-window $[a, b]$, then instead of executing X at a, we execute it at $\frac{a+b}{2}$. This enables RTE* to adapt to unexpected durations that fall outside the STNU's fixed bounds.

5 Empirical Evaluation

We evaluated the robust execution of PSTNs by generating random PSTN instances, then executing them using the RTE* algorithm based on the approximating STNU, converted to a dispatchable ESTNU. We randomly generated durations for the probabilistic links according to their distributions. Since the probabilistic durations could fall outside the contingent bounds of the ESTNU, RTE* might not succeed in all instances, but the percentage of successful executions across random trials provides a measure of the PSTN's robustness.

We wanted to evaluate whether (1) creating a dynamically controllable STNU to approximate a PSTN; and (2) taking advantage of the flexibility offered by the RTE* execution algorithm might lead to a greater percentage of successful PSTN executions, even in cases

[6] A projection of an ESTNU is the STN derived from forcing its contingent durations to take on fixed values. Each edge in an ESTNU projects onto an ordinary STN edge. For example, in the projection where $C - A = 4$, the edges $(A, c{:}2, C)$, $(C, C{:}{-9}, A)$, $(W, C{:}{-7}, A)$ and $(V, C{:}{-3}, A)$ project onto the ordinary edges $(A, 4, C)$, $(C, -4, A)$, $(W, -4, A)$ and $(V, C{:} -3, A)$, respectively [18].

Table 1 Results using `genApproxSTNU` to generate DC approximating STNUs for PSTNs

#PSTNs	n	k	$\overline{m}$	exTime [s]	optTime [s]	#NLOprobs	%probMass	#RCs
24	500	50	1558	0.191	0.141	0.96	77	11
24	1000	100	3136	0.223	0.042	1.00	67	17
14	1500	150	4713	0.573	0.100	1.21	43	10
17	2000	200	6289	0.914	0.046	1.11	53	16

Table 2 Results of RTE* execution algorithm on PSTNs: Earliest-First (EF) vs. *Midpoint (MP)*

#PSTNs	n	k	$\overline{m}$	execTP (μs)	%trials-in succ		%trials-out succ		%trials-out fail		num out if succ		num out if fail	
					EF	*MP*	EF	*MP*	EF	*MP*	EF	*MP*	EF	*MP*
24	500	50	2500	9.16	73	*73*	5	*5*	22	*22*	1.08	*1.09*	1.06	*1.06*
24	1000	100	5119	14.98	66	*66*	8	*6*	26	*28*	1.03	*1.03*	1.10	*1.12*
14	1500	150	7883	26.14	58	*58*	4	*7*	38	*35*	1.07	*1.08*	1.16	*1.15*
17	2000	200	106522	31.05	53	*53*	8	*8*	39	*39*	1.10	*1.11*	1.21	*1.23*

where the sampled durations fall outside the STNU's fixed bounds. Toward that end, we took *non-DC* STNUs from a published benchmark [25] and converted them into PSTNs as described in the Appendix (cf. the `GenPSTN` algorithm, Algorithm 4). The results of this phase are summarized in Table 1, where n, k, and m are the numbers of timepoints, contingent durations, and constraints; "exTime" is the average time to execute `genApproxSTNU`; "optTime" is the average time spent running the non-linear optimization solver; "#NLOprobs" is the average number of calls to the non-linear optimization solver; "%probMass" is the average probability mass of the probabilistic links captured by the approximating STNU; and "#RCs" is the number of approximating STNUs having one or more activation timepoints participating in rigid components. As expected, the percentage of the probability mass captured by the approximating STNU fell as the number of contingent durations increased since, for example, $.995^{50} \approx .778$, whereas $.995^{200} \approx .367$. In addition, since the initial STNU was non-DC, making it DC could require reducing contingent ranges significantly.

After converting the STNU instances into their minimal dispatchable form [17], we ran the RTE* algorithm 200 times on each dispatchable ESTNU, where the contingent durations were obtained by randomly sampling the associated log-normal distributions (15800 executions in total). To test the impact of the execution strategy on the rate of successful execution, the execution of each network in the same situation (i.e., in the same projection) was run twice: once with the earliest-first strategy, which executes timepoints as soon as possible, and once with the midpoint strategy, which executes timepoints at the midpoints of their time-windows. Table 2 summarizes our results, where: "execTP" reports the average time (in μsecs) to schedule each timepoint; "%trials-in succ", the percentage of executions/trials where all sampled durations fell within the respective contingent bounds of the ESTNU. For such cases, the execution strategy (earliest-first results in plain text, midpoint in italic) is irrelevant because any RTE* execution is guaranteed to succeed for dispatchable ESTNUs. Column "%trials-out succ" reports the percentage of trials where one or more contingent durations fell outside the ESTNU's contingent bounds (called *outlier trials*), but the execution succeeded anyway due to the flexibility of RTE* (higher value represents the best performance); while "%trials-out fail" reports the average number of outlier trials where the execution failed (lower value represents the best performance). Column "num out if fail" reports the average number of outlier durations in failed executions; while "num out if succ" reports the average number of outlier durations in successful executions. The comparison of the "%probMass" values from Table 1 and "%trials-in succ" from Table 2 confirms that the probability mass captured by the ESTNU's contingent links corresponds to situations that always generate successful executions. It is not clear if the execution strategy for the controllable timepoints (earliest-first or midpoint) can increase the rate of successful executions, given that the number of different PSTNs is limited. Further investigation is necessary, including on PSTNs from real-world applications. Nonetheless, our results provide evidence that the RTE* algorithm makes it possible to have successful executions even when one or more contingent durations are outside the ESTNU's bounds.

Our implementations are publicly available [26].

6 Conclusions

The paper presented a new approach to the robust execution of PSTNs that takes advantage of several recent efficient algorithms for:

1. finding and compactly representing SRN cycles in non-DC STNUs;
2. converting DC STNUs into equivalent, dispatchable ESTNUs having a minimal number of edges; and
3. flexibly and efficiently executing ESTNUs in real time.

We presented a new algorithm to generate an approximating STNU that aims to maximize the combined probability mass of the PSTN's probabilistic durations while maintaining the dynamic controllability of the STNU; and a formal analysis of SRN cycles that provided new insights into how to efficiently resolve them while avoiding issues arising in past approaches. Our empirical evaluation of our approach provides evidence of its effectiveness on robustly executing PSTNs derived from a publicly available benchmark. In particular, it shows that approximating a PSTN by a dispatchable ESTNU and taking advantage of a flexible real-time execution algorithm can increase the chances for a successful execution of that PSTN.

References

1 Nikhil Bhargava. Multi-Agent Coordination under Uncertain Communication. *33rd AAAI Conference on Artificial Intelligence (AAAI-19)*, 33(1):9878–9879, 2019. doi:10.1609/aaai.v33i01.33019878.

2 Massimo Cairo, Luke Hunsberger, and Romeo Rizzi. Faster Dynamic Controllablity Checking for Simple Temporal Networks with Uncertainty. In *25th International Symposium on Temporal Representation and Reasoning (TIME-2018)*, volume 120 of *LIPIcs*, pages 8:1–8:16, 2018. doi:10.4230/LIPIcs.TIME.2018.8.

3 Rosy Chen, Yiran Ma, Siqi Wu, and James C. Boerkoel, Jr. Sensitivity analysis for dynamic control of pstns with skewed distributions. In *33rd International Conference on Automated Planning and Scheduling (ICAPS 2023)*, volume 33, pages 95–99, 2023. doi:10.1609/icaps.v33i1.27183.

4 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 4th Edition*. MIT Press, 2022. URL: https://mitpress.mit.edu/9780262046305/introduction-to-algorithms.

5 Rina Dechter, Itay Meiri, and J. Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49(1-3):61–95, 1991. doi:10.1016/0004-3702(91)90006-6.

6 Maya Abo Dominguez, William La, and James C. Boerkoel Jr. Modeling human temporal uncertainty in human-agent teams. *CoRR*, abs/2010.04849, 2020. URL: https://arxiv.org/abs/2010.04849, arXiv:2010.04849.

7 Cheng Fang, Peng Yu, and Brian C. Williams. Chance-constrained probabilistic simple temporal problems. In *28th AAAI Conference on Artificial Intelligence (AAAI-2014)*, volume 3, pages 2264–2270, 2014. doi:10.1609/aaai.v28i1.9048.

8 Michael Gao, Lindsay Popowski, and James C. Boerkoel, Jr. Dynamic Control of Probabilistic Simple Temporal Networks. In *34th AAAI Conference on Artificial Intelligence (AAAI-20)*, volume 34, pages 9851–9858, 2020. `doi:10.1609/aaai.v34i06.6538`.

9 Philip E. Gill, Walter Murray, and Michael A. Saunders. Snopt: An sqp algorithm for large-scale constrained optimization. *SIAM Review*, 47(1):99–131, 2005. `doi:10.1137/S0036144504446096`.

10 Luke Hunsberger. Fixing the semantics for dynamic controllability and providing a more practical characterization of dynamic execution strategies. In *16th International Symposium on Temporal Representation and Reasoning (TIME-2009)*, pages 155–162, 2009. `doi:10.1109/TIME.2009.25`.

11 Luke Hunsberger. Magic Loops in Simple Temporal Networks with Uncertainty–Exploiting Structure to Speed Up Dynamic Controllability Checking. In *5th International Conference on Agents and Artificial Intelligence (ICAART-2013)*, volume 2, pages 157–170, 2013. `doi:10.5220/0004260501570170`.

12 Luke Hunsberger. Magic Loops and the Dynamic Controllability of Simple Temporal Networks with Uncertainty. In Joaquim Filipe and Ana Fred, editors, *Agents and Artificial Intelligence*, volume 449 of *Communications in Computer and Information Science (CCIS)*, pages 332–350, 2014. `doi:10.1007/978-3-662-44440-5_20`.

13 Luke Hunsberger. Efficient execution of dynamically controllable simple temporal networks with uncertainty. *Acta Informatica*, 53(2):89–147, 2015. `doi:10.1007/s00236-015-0227-0`.

14 Luke Hunsberger and Roberto Posenato. Speeding up the RUL^- Dynamic-Controllability-Checking Algorithm for Simple Temporal Networks with Uncertainty. In *36th AAAI Conference on Artificial Intelligence (AAAI-22)*, volume 36-9, pages 9776–9785. AAAI Pres, 2022. `doi:10.1609/aaai.v36i9.21213`.

15 Luke Hunsberger and Roberto Posenato. A Faster Algorithm for Converting Simple Temporal Networks with Uncertainty into Dispatchable Form. *Information and Computation*, 293(105063):1–21, 2023. `doi:10.1016/j.ic.2023.105063`.

16 Luke Hunsberger and Roberto Posenato. A Faster Algorithm for Finding Negative Cycles in Simple Temporal Networks with Uncertainty. In *The 31st International Symposium on Temporal Representation and Reasoning (TIME-2024)*, volume 318 of *LIPIcs*, 2024. `doi:10.4230/LIPIcs.TIME.2024.8`.

17 Luke Hunsberger and Roberto Posenato. Converting Simple Temporal Networks with Uncertainty into Minimal Equivalent Dispatchable Form. In *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2024)*, volume 34, pages 290–300, 2024. `doi:10.1609/icaps.v34i1.31487`.

18 Luke Hunsberger and Roberto Posenato. Foundations of Dispatchability for Simple Temporal Networks with Uncertainty. In *16th International Conference on Agents and Artificial Intelligence (ICAART 2024)*, volume 2, pages 253–263. SCITEPRESS, 2024. `doi:10.5220/0012360000003636`.

19 Erez Karpas, Steven J. Levine, Peng Yu, and Brian C. Williams. Robust Execution of Plans for Human-Robot Teams. In *25th Int. Conf. on Automated Planning and Scheduling (ICAPS-15)*, volume 25, pages 342–346, 2015. `doi:10.1609/icaps.v25i1.13698`.

20 Dimitri Kececioglu et al. *Reliability Engineering Handbook*, volume 1. DEStech Publications, Inc, 2002.

21 Paul Morris. A Structural Characterization of Temporal Dynamic Controllability. In *Principles and Practice of Constraint Programming (CP-2006)*, volume 4204, pages 375–389, 2006. `doi:10.1007/11889205_28`.

22 Paul Morris. Dynamic controllability and dispatchability relationships. In *Int. Conf. on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR-2014)*, volume 8451 of *LNCS*, pages 464–479. Springer, 2014. `doi:10.1007/978-3-319-07046-9_33`.

23 Paul Morris, Nicola Muscettola, and Thierry Vidal. Dynamic control of plans with temporal uncertainty. In *17th Int. Joint Conf. on Artificial Intelligence (IJCAI-2001)*, volume 1, pages 494–499, 2001. URL: https://www.ijcai.org/Proceedings/01/IJCAI-2001-e.pdf.

24 Nicola Muscettola, Paul H. Morris, and Ioannis Tsamardinos. Reformulating temporal plans for efficient execution. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning*, KR'98, page 444–452, 1998.

25 Roberto Posenato. STNU Benchmark version 2020, 2020. Last access 2022-12-01. URL: https://profs.scienze.univr.it/~posenato/software/cstnu/benchmarkWrapper.html.

26 Roberto Posenato. CSTNU Tool: A Java library for checking temporal networks. *SoftwareX*, 17:100905, 2022. doi:10.1016/j.softx.2021.100905.

27 Ioannis Tsamardinos. A probabilistic approach to robust execution of temporal plans with uncertainty. In *Methods and Applications of Artificial Intelligence (SETN 2002)*, volume 2308 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 97–108, 2002. doi:10.1007/3-540-46014-4_10.

28 Ioannis Tsamardinos, Nicola Muscettola, and Paul Morris. Fast Transformation of Temporal Plans for Efficient Execution. In *15th National Conf. on Artificial Intelligence (AAAI-1998)*, pages 254–261, 1998. URL: https://cdn.aaai.org/AAAI/1998/AAAI98-035.pdf.

29 Thierry Vidal and Hélène Fargier. Handling contingency in temporal constraint networks: from consistency to controllabilities. *J. of Experimental & Theoretical Artificial Intelligence*, 11(1):23–45, 1999. doi:10.1080/095281399146607.

30 Andrew Wang and Brian C. Williams. Chance-Constrained Scheduling via Conflict-Directed Risk Allocation. In *29th Conference on Artificial Intelligence (AAAI-2015)*, volume 29, 2015. doi:10.1609/aaai.v29i1.9693.

31 Andrew J. Wang. *Risk-bounded Dynamic Scheduling of Temporal Plans*. PhD thesis, Massachusetts Institute of Technology, 2022. URL: https://hdl.handle.net/1721.1/147542.

32 Peifeng Yin, Ping Luo, Wang-Chien Lee, and Min Wang. Silence is also evidence: interpreting dwell time for recommendation from psychological perspective. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 989–997, 2013. doi:10.1145/2487575.2487663.

33 Peng Yu, Cheng Fang, and Brian Charles Williams. Resolving uncontrollable conditional temporal problems using continuous relaxations. In *24th International Conference on Automated Planning and Scheduling, ICAPS 2014*. AAAI, 2014. doi:10.1609/icaps.v24i1.13623.

Appendix

A Procedure for Tighten Contingent Bounds to Resolve an SRN

In this section, we propose `nlpOpt`, a possible algorithm that tightens contingent bounds to resolve an SRN cycle using "Sparse Nonlinear OPTimizer" (SNOPT) library [9]. SNOPT is a software package for solving large-scale optimization problems (linear and nonlinear programs). It employs a sparse Sequential quadratic programming (SQP) algorithm with limited-memory quasi-Newton approximations to the Hessian of Lagrangian.

In `nlpOpt` we assume that the bounds on contingent links are monotonically tightened, using only a single linear constraint per iteration. A different possibility is to collect the linear constraints from each iteration and run the optimization solver on all of the accumulated constraints.

In the experimental evaluation, it was not possible to use the SNOPT library due to a compatibility problem. MatLab-Optimization Toolbox library offers the `fmincon` function to solve *minimization constrained nonlinear problems* using a sparse Sequential quadratic programming (SQP) algorithm, the same technique used by SNOPT. Therefore, we adapted

Algorithm 3 The `nlpOpt` algorithm: tighten contingent bounds to resolve an SRN cycle

Input: k: the number of contingent durations; $(\mu_1, \ldots, \mu_k)$ and $(\sigma_1, \ldots, \sigma_k)$: k-vectors of μ and σ parameters for log-normal distributions; `coeffs` and `const`: a matrix of coefficients and a corresponding vector of lower bounds for one or more linear constraints; $(x_1, y_1, \ldots, x_k, y_k)$: a vector of initial bounds for the contingent durations

Output: $(\mathtt{v}, \mathtt{F})$, where `v` contains optimized bounds for the k contingent durations, and `F` is the corresponding value of the objective function

$\mathtt{snN} := 2k$

$\mathtt{numCs} := \mathtt{numRows}(\mathtt{coeffs})$ // `numCs` is the number of linear constraints

$\mathtt{snNF} := 1 + \mathtt{numCs}$ // `snNF` includes 1 for the objective function

`F` := a new vector with `snNF` slots // `F` will hold values of objective function and linear constraints

// Initialize `v`, the vector of variables

$\mathtt{v} := (x_1, y_1, \ldots, x_k, y_k)$

// Set lower and upper bounds for the variables in `v`

$\mathtt{vlow} := (x_1, e^{\mu_1}, x_2, e^{\mu_2}, \ldots, x_k, e^{\mu_k})$

$\mathtt{vupp} := (e^{\mu_1}, y_1, e^{\mu_2}, y_2, \ldots, e^{\mu_k}, y_k)$

// Set lower and upper bounds for the objective function (in $[-1, 0]$) and the linear constraints

$\mathtt{Flow} := (-1, \mathtt{const}[1], \mathtt{const}[2], \ldots, \mathtt{const}[\mathtt{numCs}])$

$\mathtt{Fupp} := (0, \infty, \infty, \ldots, \infty)$

// Local function that SNOPT uses to compute the objective function and linear constraints

Function `stnuUsrFun(`v`):` // $\mathtt{v} = (\ell_1, u_1, \ldots, \ell_k, u_k)$

    // Store the value of the objective function in $\mathtt{F}[1]$

    $\mathtt{F}[1] := \Pi_{1 \leq i \leq k} \left(\mathtt{lnCDF}(u_i, \mu_i, \sigma_i) - \mathtt{lnCDF}(\ell_i, \mu_i, \sigma_i)\right)$ // $\mathtt{lnCDF}$ = log-normal CDF

    // Store the values of the lefthand sides of the linear constraints in $\mathtt{F}[2], \ldots, \mathtt{F}[\mathtt{snNF}]$

    foreach $j \in \{1, \ldots, \mathit{numRows}(\mathit{coeffs})\}$ **do**

        $\mathtt{F}[j+1] := \mathtt{coeffs}[j][1] * v[1] + \ldots \mathtt{coeffs}[j][2k] * \mathtt{v}[2k]$

// Call the SNOPT solver, which destructively modifies `v`

$(\mathtt{v}, \mathtt{F}, \ldots) := \mathtt{snSolveA}(\mathtt{v}, \mathtt{vlow}, \mathtt{vupp}, \mathtt{Flow}, \mathtt{Fupp}, \&\mathtt{stnuUsrFun})$

return $(\mathtt{v}, \mathtt{F})$

Algorithm 4 The `GenPSTN` algorithm: generation of a PSTN candidate from an STNU

Input: $\mathcal{N} = (\mathcal{T}_N, \mathcal{C}_N, \mathcal{L})$: an STNU where $\mathcal{L}$ is a set of k contingent links, each of the form (A_i, x_i, y_i, C_i), where $A, C \in \mathcal{T}$ and $0 < x < y < \infty$.

Output: $\mathcal{S} = (\mathcal{T}_S, \mathcal{C}_S, \mathcal{M})$: a PSTN where $\mathcal{M} = \{(A_i, C_i, \text{Lognormal}(\mu_i, \sigma_i)) \mid \in \{1, \ldots, k\}\}$

$\mathcal{T}_S := \mathcal{T}_N$

$\mathcal{C}_S := \mathcal{C}_N$

$\mathcal{M} := \emptyset$

$\sigma_f := 0.3$ // Factor to limit the final σ value

foreach $(A, x, y, C) \in \mathcal{L}$ **do**

    $M = (x + y)/2$

    $S = \sigma_f (y - x)/2$

    $\mu = \ln(M^2 / \sqrt{M^2 + S^2})$

    $\sigma = \sqrt{\ln(1 + S^2/M^2)}$

    $\mathcal{M} := \mathcal{M} \cup \{(A, C, \text{Lognormal}(\mu, \sigma))\}$

return $(\mathcal{T}_S, \mathcal{C}_S, \mathcal{M})$

the `nlpOpt` algorithm, reformulating the optimization problem as a minimization one and using a MatLab script to represent the non-linear objective function.

B PSTN Generation

To generate a set of PSTN instances for our benchmark, we considered the set of random *non-DC* STNUs from a published benchmark [25]. Such instances aim to represent the temporal representation of business processes organized in worker lanes. Contingent links represent tasks and ordinary links represent temporal deadlines or release times of such tasks.

Each random STNU was converted into a PSTN using the `GenPSTN` algorithm described in Algorithm 4. For each contingent link (A, x, y, C) in the STNU, `GenPSTN` creates a probabilistic duration with a log-normal distribution with parameters μ and σ chosen to ensure that the mean of the distribution is $(x + y)/2$, and three standard deviations captures the entire range $[x, y]$ [20]. Starting with a non-DC STNU guarantees that the initial STNU candidate generated by `genApproxSTNU` would not be DC and, hence, would require multiple iterations to find an approximating STNU that was DC. However, because some non-DC STNUs have negative cycles comprising only ordinary edges and, hence, cannot be made DC by restricting their contingent ranges, only the PSTNs for which DC approximating STNUs can be created were kept.