Delay Controllability: Multi-Agent Coordination under Communication Delay

Nikhil Bhargava nkb@mit.edu
Christian Muise cjmuise@mit.edu
Tiago Vaquero tvaquero@mit.edu
Brian Williams williams@mit.edu
Model-based Embedded and Robotics Systems
MIT, Cambridge, MA 02139 USA

Abstract

Simple Temporal Networks with Uncertainty provide a useful framework for modeling temporal constraints and, importantly, for modeling actions with uncertain durations. To determine whether we can construct a schedule for a given network, we typically consider one of two types of controllability: dynamic or strong. These controllability checks have strict conditions on how uncertainty is resolved; uncertain outcomes are either recognized immediately or not at all. In this paper, we introduce *delay controllability*, a novel generalization of both strong and dynamic controllability that additionally exposes a large range of controllability classes in between. To do so, we use a delay function γ to parameterize our controllability checking. This delay function represents the difference between when an event happens and the time that it is observed. We also provide a single unified algorithm for checking delay controllability that runs in $O(n^3)$ time, matching the best known runtime for dynamic controllability, which we use to motivate the decision to generalize dynamic and strong controllability. We conclude by providing an empirical evaluation of delay controllability, demonstrating its superior accuracy and practical efficiency as compared to other existing approximations.

1. Introduction

In temporal planning, an agent is given a set of events and temporal constraints and must determine whether it is possible to construct a schedule for those events such that all constraints are satisfied. When the durations of all actions are within the agent's control, determining this is relatively straightforward. However, it is much more difficult to plan for events whose durations cannot be precisely controlled. These kinds of events are commonplace in daily life and can include things like how long it takes to drive to work or when it will start to rain. We say that a collection of events and temporal constraints is controllable if there is some way to guarantee the success of a plan despite this uncertainty.

Historically, two types of controllability have been most effective for modeling scheduling problems: dynamic and strong controllability (Vidal & Fargier, 1999). In this paper, we introduce a single parent definition which is expressive enough to unite the two of them, *delay controllability*. Delay controllability is parameterized by a delay function γ that we use to model the difference between when an uncontrollable event occurs and when an agent can observe and react to it.

Consider the difficulty of scheduling events in a scenario where multiple agents have to coordinate with one another. From the perspective of any one individual agent, the actions of other agents may be bounded by temporal constraints, but the actual time taken for any given action is highly uncertain. If all agents are in constant communication with one another, then modeling the creation of a schedule under this uncertainty reduces to dynamic controllability, as all uncertainty is resolved as soon as individual actions are completed. However, this is not a realistic portrayal of multi-agent coordination. In reality, agents might only relay the relevant information after some indeterminate delay or may never provide any information at all. Strong controllability scheduling techniques give valid solutions, when they work, but too often they are overly conservative since they assume that external uncertainty is never resolved. Delay controllability provides a way to create robust schedules that make use of the full flexibility available.

Because delay controllability is a generalization of the principles of dynamic and strong controllability, we can build on the shoulders of previously established research to apply familiar concepts while retaining the runtime complexity and efficiency of the best of these algorithms. In this paper in particular, we will extend and generalize a set of constraint derivation rules provided by (Morris & Muscettola, 2005), adapt a state-of-the-art dynamic controllability checking algorithm to handle delay controllability (Morris, 2014), and demonstrate correctness by modifying a known dynamic controllability execution strategy and its proof of correctness (Hunsberger, 2016).

Beyond building a sound and complete delay controllability checker, there are many interesting bodies of work that could be extended to incorporate delay controllability. Conditional Simple Temporal Networks with Uncertainty extend the temporal network model by allowing conditional enforcement of constraints based on the observation of pre-specified events and use dynamic controllability to determine appropriate execution strategies (Combi, Hunsberger, & Posenato, 2013). For more specialized use cases, iterative dynamic controllability checks show improvements over non-iterative solutions (Stedl & Williams, 2005; Shah, Stedl, Williams, & Robertson, 2007; Nilsson, Kvarnström, & Doherty, 2013, 2014; Bhargava, Vaquero, & Williams, 2017). While we do not cover how to extend delay controllability to these concepts in this work, we believe that such extensions are well within reach.

The specific contributions of this work are threefold. Our primary contribution is theoretical in nature. We provide a generalized definition of controllability, namely delay controllability, and show how it is expressive enough to capture the semantics of the most widely used notions of controllability. In particular, our generalization captures both strong and dynamic controllability which were heretofore thought to be entirely distinct concepts. Our new definition also opens the door for a large number of intermediate controllability classes that can be used to model temporal problems in higher fidelity.

Second, we provide an efficient sound and complete algorithm for checking delay controllability, which demonstrates that our generalization does not force us to sacrifice worst-case runtime performance. Our approach gives us a single $O(n^3)$ algorithm that is capable of checking either strong or dynamic controllability (as well as everything in between), reinforcing the notion that these two concepts are different incarnations of the same idea. Dynamic controllability is known to be computable in $O(n^3)$ time (Morris, 2014), meaning our approach is competitive with state-of-the-art dynamic controllability checkers. Strong

controllability checking represents a special case of the delay controllability problem, and in general can be reduced to negative cycle detection (Vidal & Fargier, 1999), which has a best known strongly polynomial runtime of $O(mn)$.

Finally, we conclude by providing an empirical evaluation of delay controllability. We show that attempting to capture the nuances of delay controllability using dynamic or strong controllability as an approximation leads to frequent errors, and we also demonstrate that delay controllability is fast enough in practice for use as a subroutine in other larger systems.

The rest of the paper proceeds as follows. In section 2, we introduce the necessary background and definitions needed for the delay controllability formalism. In section 3, we formally define delay controllability and show how it differs from other related concepts. In section 4, we introduce a graphical representation for our temporal networks and show how to use it to derive and enforce new constraints. Section 5 presents the main algorithm that efficiently explores that graphical representation, and section 6 provides the proof showing that our algorithm efficiently verifies delay controllability. In section 7, we walk through an empirical evaluation of delay controllability, both in terms of the quality of the model and in terms of its practical efficiency, and we conclude with a summary of our work in section 8.

2. Background & Definitions

Dechter, Meiri, and Pearl (1991) introduced Simple Temporal Networks (STNs) as a way to formally model temporal constraints. An STN is represented as a graph where the nodes represent timepoints and the links represent binary constraints between timepoints (e.g., event A must happen at least 30 minutes after event B; event B must happen no more than 15 minutes after event C). We say that an STN is *feasible* if there is an assignment of values to timepoints such that all constraints are satisfied.

As described, however, STNs are incapable of modeling actions whose durations cannot be precisely controlled by the agent. In multi-agent problems in particular, many events are explicitly outside of the agent's control. Temporal uncertainty is a natural part of scheduling and any formalism concerned with scheduling must take it into account. Simple Temporal Networks with Uncertainty (STNUs) give us a way to model these uncontrollable actions by categorizing timepoints as either *activated* or *received* and categorizing constraints as either *free* or *contingent* (Vidal & Fargier, 1999).

Definition 1. STNU (Vidal & Fargier, 1999)
An STNU is a 4-tuple $\langle X_b, X_e, R_c, R_g \rangle$ where:

- X_b is the set of activated timepoints
- X_e is the set of received timepoints
- R_c is the set of free constraints of the form $l_c \leq x_i - x_j \leq u_c$, where $x_i, x_j \in X_b \cup X_e$
- R_g is the set of contingent constraints of the form $l_g \leq e_i - b_j \leq u_g$, where $e_i \in X_e$, $b_j \in X_b$

Activated timepoints are events whose times are assigned by an agent while received timepoints are assigned values by some external actor. Free constraints behave like constraints in an STN and are free to relate any two timepoints. In contrast, contingent

constraints represent actions whose durations cannot be precisely controlled. They relate a starting activated timepoint to an ending received timepoint. By convention, we say that all contingent constraints have non-negative bounds since a received timepoint must happen after the activated timepoint that caused it. Free constraint bounds can take on any sign.

Since received timepoints have unknown assignments in an STNU, it is difficult to reason directly about the feasibility of an STNU. Instead we evaluate whether an STNU is *controllable*, or whether it is possible to provide an assignment to all activated timepoints in response to some observation of received timepoints. Historically, STNU controllability has come in three forms: weak, dynamic, and strong (Vidal & Fargier, 1999). In this paper, we will restrict our focus to the latter two, as strong and dynamic controllability are useful notions in scheduling execution of an STNU and weak controllability tends to be less so.

Informally, we say that an STNU is *dynamically controllable* if it is possible to just-in-time assign values to timepoints given knowledge about assignments to only those timepoints that happened in the past. An STNU is *strongly controllable* if there exists a single set of assignments to all activated timepoints, such that all temporal constraints are satisfied regardless of the assignment of durations to contingent constraints. To illustrate the difference between the concepts consider the following example:

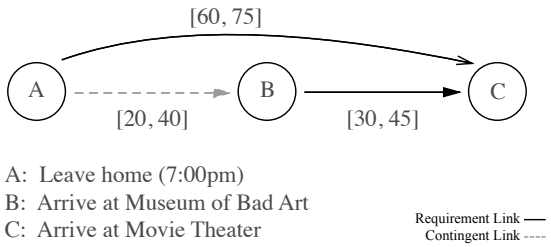

Figure 1: Sam goes to the Museum of Bad Art and the movies. Figure for Example 1.

Example 1. Sam goes to the Museum of Bad Art and the movies. See Figure 1.

> Sam wants to spend between 30 and 45 minutes at the Museum of Bad Art and then attend a movie at the Somerville Theater right upstairs at 8:15pm. It is now 7:00pm, and it will take her between 20 and 40 minutes to drive to the museum. She does not want to be more than 15 minutes early to the movie, and she definitely does not want to be late.

Example 1 is not strongly controllable in that there is no way to pre-commit to a schedule. If Sam decides to head upstairs from the museum at any time before 8:10pm and it takes 40 minutes to drive to the museum, then she will not spend enough time at the museum. In contrast, if she decides to head upstairs after 8:10pm and arrives at the museum in 20 minutes, she will spend too much time at the Museum of Bad Art. However, it is possible to construct a schedule for this problem on the fly, meaning the scenario is dynamically controllable. When Sam arrives at the museum, she can decide on times that respect all the constraints. If she gets there in under 30 minutes, she can head up to the

theater at 8:00pm, but if it takes her longer, she can head upstairs at 8:10pm. With a slight modification to problem's constraints, we can generate an example that is neither strongly nor dynamically controllable.

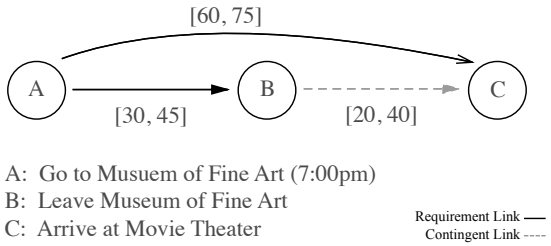

Figure 2: Sam goes to the Museum of Fine Art and the movies. Figure for Example 2.

Example 2. Sam goes to the Museum of Fine Art and the movies. See Figure 2.

> Sam lives next door to the Museum of Fine Art and wants to spend between 30 and 45 minutes at the museum before going to see a movie at the Somerville Theater at 8:15pm. It is now 7:00pm, and it will take between 20 and 40 minutes to drive from the museum to the theater. She does not want to be more than 15 minutes early to the movie and definitely does not want to be late.

In contrast to the previous example, Example 2 is not dynamically controllable. At the start, Sam does not know how long her commute will be, so if she spends more than 35 minutes at the museum, she may miss the movie if her commute takes 40 minutes. In contrast, if she spends less than 35 minutes there, she may be too early if her commute takes just 20 minutes.

Instead of performing ad hoc analysis on our examples to determine their controllability, we want to take a more principled approach to constructing a schedule for our temporal problems. In order to robustly determine whether an STNU is controllable, we generally follow the same formula. First, we generate a candidate *assignment* of values to activated timepoints.

Definition 2. Assignment (Vidal & Fargier, 1999)
δ is a set of tuples of activated timepoints $x_b \in X_b$ and their assigned time values. For convenience, we can either say $(x_b, t) \in \delta$ or $\delta(x_b) = t$. Since all constraints are relative, without loss of generality we can assume that the minimum value assigned to any timepoint is 0. We say that $\delta_{\leq t}$ is the set of executed assignments to activated timepoints such that $\delta_{\leq t} = \{(x_b, \delta(x_b)) \text{ s.t. } \delta(x_b) \leq t\}$. For clarity, we assume that execution strategies may make decision instantaneously.

After initially picking an assignment, we may have to change our decision. Contingent constraints are inherently uncertain, but as time passes, we learn more information about the true values of those constraints. We can use what we learn to update our beliefs about the possible *situation* we are in.

Definition 3. Observed Situation (Vidal & Fargier, 1999)
Let Ω be the set of all possible sets of assignments of durations to contingent constraints. The set of all possibly observed situations before t with respect to some δ, ω is $\Omega_{\leq t,\delta,\omega} = \{\omega' \in \Omega : \{\omega_{ij} \in \omega : \delta(b_i) + \omega_{ij} \leq t\} \subseteq \omega'\}$.

Given a restricted set of possible situations based on what we have already observed, the problem of evaluating our chosen assignment becomes easier. In order to validate that our chosen assignment satisfies our STNU, we have to evaluate the STNU with respect to the true durations of all contingent constraints. Here, we use the *projection* of the STNU with respect to some set of assignments to durations of contingent constraints to verify whether our assignment satisfies the underlying problem.

Definition 4. Projection (Vidal & Fargier, 1999)
Let ω be a set of assignments of durations to contingent constraints $r_g \in R_g$. For an STNU S, we say that the projection S_ω is the STNU yielded by replacing all contingent constraints r_g with a free constraint with lower and upper bounds ω_g. S_ω has no contingent constraints, so we can treat it like an STN and assess its feasibility directly.

The choice of assignments and our ability to evolve them as we incorporate more information forms the basis for how we check controllability. The specific ways in which we choose our assignments and observe the evolution of possible situations are what dictate the type of controllability we use.

3. Generalizing the Model

Existing characterizations of the two main types of controllability, dynamic and strong, are overly restrictive. What we need instead is a solution that lies somewhere in the middle, faithfully modeling the problem of scheduling in the face of delays in observations of communication events without being overly conservative and ignoring possible solutions. Delay controllability strikes at the heart of this problem, generalizing the two forms of controllability and exposing a class of controllability problems that lie in between.

3.1 Motivating Example

In order to motivate the value of delay controllability, we introduce a pair of examples that have the same constraints restricting the actions and choices of individual agents but differ in when information is relayed.

Example 3. Sam and Alex go to the movies, but first Sam's phone needs to be recharged. See Figure 3.

> Sam wants to spend between 30 and 45 minutes at the Museum of Bad Art and then attend a movie with her friend Alex at the Somerville Theater right upstairs. After looking at the movie times, she has decided to attend the movie showing at 8:15pm. It is now 7:00pm, and it will take her between 20 and 40 minutes to drive to the museum. She does not want to be more than 15 minutes early to the movie, and she definitely does not want to be late.

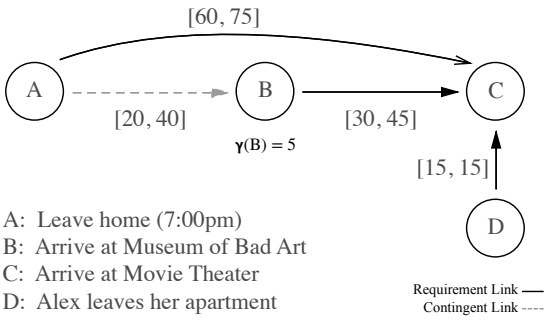

Figure 3: Sam and Alex go to the movies, but Sam's phone needs to recharge for 5 minutes to let her call Sam. Figure for Example 3.

Alex's apartment is a 15-minute walk from the Somerville Theater, so she has asked Sam to give her a call when she should leave for the movies. Unfortunately, Sam's phone needs to be recharged, and she is unable to immediately call Alex. When Sam arrives at the museum, she will be able to leave her phone to charge at the museum front desk for 5 minutes before calling Alex.

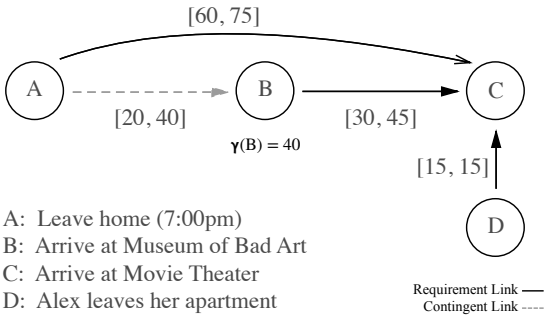

Figure 4: Sam and Alex go to the movies, but Sam's phone needs to recharge for 40 minutes to let her call Sam. Figure for Example 4.

Example 4. Sam and Alex go to the movies, but Sam's phone has been deeply discharged. See Figure 4.

Sam wants to spend between 30 and 45 minutes at the Museum of Bad Art and then attend a movie with her friend Alex at the Somerville Theater right upstairs. After looking at the movie times, she has decided to attend the movie showing at 8:15pm. It is now 7:00pm, and it will take her between 20 and 40 minutes to drive to the museum. She does not want to be more than 15 minutes early to the movie, and she definitely does not want to be late.

> Alex's apartment is a 15-minute walk from the Somerville Theater, so she has asked Sam to give her a call when she should leave for the movies. Unfortunately, Sam's phone is deeply discharged, and she is unable to immediately call Alex. When Sam arrives at the museum, she will be able to leave her phone to charge at the museum front desk for 40 minutes at which point it will power on and she can call Alex.

Example 3 is clearly controllable. As was the case in Example 1, Sam can plan on the fly to ensure that she spends an appropriate amount of time at the Museum of Bad Art while getting to the movie on time. If Sam calls Alex 5 minutes after arriving, Alex has plenty of time to make it to the movie theater in time. In contrast, Example 4 is not controllable. If it takes Sam 40 minutes to drive to the museum, she will only be able to call Alex at 8:20pm, after the movie starts.

Existing controllability models do not appropriately model the subtle difference between these two examples. Neither problem instance is strongly controllable, as no one schedule will work for Sam by herself, but we can clearly construct a valid schedule for Example 3. In contrast, when we use dynamic controllability, we discard all the difficulties of dealing with information asymmetry and our model tells us that both problem instances are controllable. It is true that if Alex immediately learns when Sam arrives at the museum, then she can always make it to the movie. However, this approach fails to model the disruption of communication that makes coordination difficult in Example 4. Being able to model delay in communication is what motivates our decision to introduce delay controllability.

3.2 Delay Controllability

In order to introduce delay controllability, we first introduce the concept of a delay function, which is used to characterize exactly when an agent can observe the outcome of an uncontrollable action.

Definition 5. Delay Function
A delay function, $\gamma : X_e \rightarrow \mathbb{R}^+ \cup \{\infty\}$, takes a received timepoint and outputs the amount of time that will pass after its assignment before its value is observed and the underlying uncertainty is resolved.

We say that an STNU is delay controllable with respect to some delay function γ if it is always possible to construct a satisfying plan for the future given the events that have already been observed.

Definition 6. Observed Situation with Delay
Let Ω be the set of all possible sets of assignments of durations to contingent constraints. The set of all possibly observed situations before t with respect to some δ, ω, γ is $\Omega_{\leq t,\delta,\omega,\gamma} = \{\omega' \in \Omega : \{\omega_{ij} \in \omega : \delta(b_i) + \omega_{ij} + \gamma(e_j) \leq t\} \subseteq \omega'\}$.

Definition 7. Delay Controllability
An STNU S is delay controllable with respect to some delay function γ iff: $\forall \omega \in \Omega$, there exists an assignment δ, such that $\forall (x,t) \in \delta$, $\forall \omega' \in \Omega_{<t,\delta,\omega,\gamma}$, there exists δ', such that $\delta_{<t,\delta} \subseteq \delta'$ and δ' is an assignment that satisfies $S_{\omega'}$.

Given our formulation of delay controllability, we see that it is quite straightforward to model strong and dynamic controllability.

Definition 8. Strong Controllability
An STNU S is strongly controllable if and only if it is delay controllable with respect to delay function $\gamma(x_e) = \infty$.

Definition 9. Dynamic Controllability
An STNU S is dynamically controllable if and only if it is delay controllable with respect to delay function $\gamma(x_e) = 0$.

3.3 Related Work

While other work has made similar attempts to extend how agents react and respond to uncontrollable events in temporal networks, we argue that their efforts are materially different from our work on delay controllability.

3.3.1 POSTNUs

Previous work on mixing notions of strong and dynamic controllability have used Partially Observable STNUs (POSTNUs) (Moffitt, 2007). A POSTNU is an STNU where received timepoints are either designated to be observable by the agent or to be entirely unobservable. To say that a POSTNU is dynamically controllable is the same as saying that the corresponding STNU is strongly controllable with respect to the unobservable timepoints and dynamically controllable with respect to the rest.

More recent work has shown that checking the dynamic controllability for certain classes of POSTNUs takes $O(n^3)$ time (Bit-Monnot, Ghallab, & Ingrand, 2016). The check for dynamic controllability is a two-step process. First, the POSTNU is compiled into a related STNU which contains none of the unobservable nodes; this transformation takes linear time. Then a standard dynamic controllability checking algorithm can be used to verify the overall controllability of the POSTNU. The runtime is dominated by the dynamic controllability checking which is known to take $O(n^3)$ time.

POSTNUs offer significant flexibility from a modeling perspective and can in fact be used to model the same types of delays that are considered by delay functions when checking delay controllability of STNUs. In Figure 5, we provide an example to illustrate how one might model delay controllability using STNUs. With POSTNUs, it is also possible to model things above and beyond what can be expressed with delay controllability.

Despite the power of POSTNUs, the decision to invoke POSTNUs to model delays in communication that are prevalent in multi-agent coordination is not without tradeoffs. The expressivity of POSTNUs makes the act of verifying POSTNU controllability more difficult in general, exposing some of the comparative advantages of using delay controllability for these types of scenarios.

Currently, the most efficient POSTNU checker is only sound and complete for networks that lack *chained contingencies* (Bit-Monnot et al., 2016), where chained contingencies are defined as group of successive links $A \Rightarrow B \Rightarrow C$ where the middle node B is involved in another contingent or requirement link (see Figure 5b).

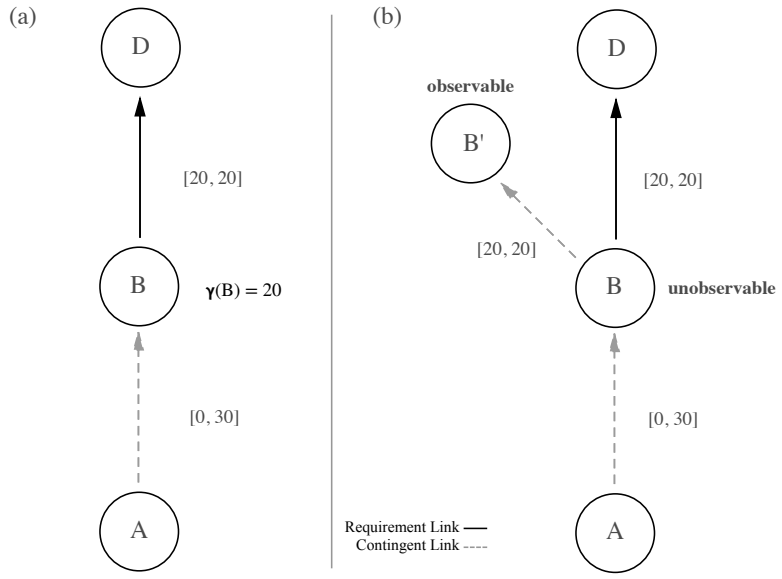

Figure 5: (a) An STNU with a contingent link that has a certain delay. (b) One possible way of rewriting the STNU as an equivalent POSTNU. This particular POSTNU exhibits a chained contingency, as B is a received timepoint that starts a contingent link and is connected to B' via a requirement link.

It is worth noting that every POSTNU that is free of chained contingencies can be represented as an STNU with a corresponding delay function. To transform a POSTNU P into STNU S with delay function γ, we only need to focus on transforming the contingent links. The biggest difference in contingent links between POSTNUs and our STNUs with delay controllability is that POSTNU contingent links may start immediately at a received timepoint, whereas our STNU-based contingent links must start at an activated timepoint. For STNUs, this limitation does not mean much in practice as the same transformation that we demonstrate here can be used to convert STNUs that allow contingent links to start at received timepoints to ones that require that all contingent links start at activated timepoints.

We divide our POSTNU contingent links based on whether they are observable or not and then further subdivide them based on whether their terminal received timepoints are immediately followed by other contingent links. For the unobservable contingent links that have no successor contingent links, we keep the original contingent link and ensure that its terminal received timepoint, x_e, respects $\gamma(x_e) = 0$. For an unobservable contingent link $A \xRightarrow{[u,v]} C$ that is followed by a successor contingent link $C \xRightarrow{[w,z]} D$, we replace the two links in our POSTNU with a new $A \xRightarrow{[u+w,v+z]} D$ whose observability is the same as the second link. From an operational perspective, this does not change the controllability of the POSTNU; because we assume our POSTNU is free of chained contingencies, C is

not involved in any other temporal constraints, and because C is unobservable, folding it into the succeeding contingent link does not affect the semantics of the network. All that remains are observable contingent links. For every observable contingent link $A \overset{[u,v]}{\Longrightarrow} C$ in the original POSTNU, we replace it with a contingent link $A \overset{[u,v]}{\Longrightarrow} C'$ and a requirement link $C' \xrightarrow{[0,0]} C$ in our STNU, where $\gamma(C') = 0$. Since C was observable in the POSTNU, we can simulate activating a timepoint immediately as it is observed, meaning that any contingent links that followed $A \overset{[u,v]}{\Longrightarrow} C$ would now start at an activated timepoint. Thus, our delay controllability framework is sufficiently capable of expressing all POSTNUs that can be efficiently checked for controllability.

While it is likely that advances can be made to expand the set of POSTNUs that can be checked efficiently, the problem remains that the way that a set of temporal constraints is encoded can affect whether controllability can be checked efficiently. For example, the transformation in Figure 5 provides an example transformation from an STNU with delay function to a POSTNU that results in a chained contingency, but it is possible to construct an equivalent POSTNU without any such chained contingencies. From a theoretical perspective, this makes the delay controllability framework a much more satisfying one to use. As we will show in the rest of our paper, any set of temporal constraints that can be expressed as an STNU with a delay function can be guaranteeably efficiently checked for controllability. With POSTNUs, in contrast, different network encodings which represent the same underlying constraints can affect whether or not it is possible to check controllability efficiently. This distinction makes delay controllability an important concept to use and build upon at least until more complete algorithms are discovered for POSTNUs.

3.3.2 ϵ-dynamic controllability

ϵ-dynamic consistency for Conditional Simple Temporal Networks is a way of validating whether an execution strategy exists for a network when agents have non-instantaneous reaction times (Comin & Rizzi, 2015; Hunsberger & Posenato, 2016). Since contingent actions are generally outside of the control of the main actor, it is reasonable to model the time it takes to react to any new information with a single parameter ϵ. It is quite natural to extend the same reasoning to STNUs to create ϵ-dynamic controllability, where we measure if an STNU is controllable subject to some agent's reaction time.

ϵ-dynamic controllability, as we have described it, provides a generalization for strong and dynamic controllability through a variation of the constant ϵ. However, ϵ-dynamic controllability is lacking in that it is unable to appropriately model mixtures of controllability; it is equivalent to delay controllability with respect to $\gamma(x_e) = \epsilon$, meaning that it cannot deal with the fact that an agent might be able to react to some events instantaneously (e.g. Sam knows immediately when she arrives at the museum but Alex only finds out 5 minutes later, as in Example 3).

In the rest of the paper, we present a unified algorithm that checks delay controllability in $O(n^3)$, which matches the best known runtime for the more specific problem of checking dynamic controllability. In addition to generalizing strong and dynamic controllability, delay

controllability generalizes controllability POSTNUs without chained contingencies as well as ϵ-dynamic controllability. Our approach for checking delay controllability is similar to the state-of-the-art algorithm for checking dynamic controllability but pays special attention to the complexity introduced with the delay function. Our argument proceeds as follows. First, we introduce a series of constraint derivation rules that allow us to propagate existing constraints and derive new ones within our STNU. Next, we provide a definition for semi-reducible negative cycles in a labeled distance graph and give an efficient algorithm for finding them. Finally, we prove that an STNU is delay controllable if and only if it is free of semi-reducible negative cycles to show that we have an efficient algorithm for determining delay controllability.

4. Constraint Derivation Rules

To simplify our analysis of STNUs, we often represent an STNU as a *labeled distance graph* (Morris & Muscettola, 2005). By using a graphical representation of the STNU, we can leverage existing graph algorithms to more efficiently explore our constraint space and verify the controllability of our network. In a labeled distance graph, all timepoints $x \in X_b \cup X_e$ are represented as nodes. Each free constraint of the form $l \leq x_B - x_A \leq u$ produces two edges. One edge $A \to B$ has weight u, and the reverse edge $B \to A$ has weight $-l$. Each contingent constraint of the form $l \leq x_C - x_A \leq u$ produces the same two edges, $A \to C$ with weight u and $C \to A$ with weight $-l$. In addition, each contingent constraint produces two additional labeled edges. Along $A \to C$, we add another edge with weight l and lower-case label c, and along $C \to A$, we add an edge with weight $-u$ and upper-case label C.

For notational convenience, we use $A \xrightarrow{[u,v]} B$ to represent a free constraint between A and B with lower bound u and upper bound v and use $A \xRightarrow{[x,y]} C$ to represent a contingent constraint between A and C with lower bound x and upper bound y. For the labeled distance graph, we use $A \xrightarrow{w} B$ to represent an unlabeled edge from A to B with weight w. We use $A \xrightarrow{c:x} C$ to represent a lower-case edge from A to C with label c and weight x and similarly use $C \xrightarrow{C:y} A$ to represent an upper-case edge from C to A with label C and weight y. See Figure 6 for an example of how an STNU can be represented as a labeled distance graph. Finally, we use $A \rightsquigarrow B$ to represent a path from A to B in the labeled distance graph.

The labeled distance graph serves as an encoding of the STNU's temporal constraints. An unlabeled edge $A \xrightarrow{w} B$ represents the inequality $x_B - x_A \leq w$. Labeled edges are used to represent conditional constraints. An edge with a lower-case label represents a constraint that must hold if the contingent constraint associated with the label were to take on its minimum value. Similarly, an edge with an upper-case label represents a constraint that must hold if the associated contingent link were to take on its maximum value.

Because each edge maps back to a constraint, we can use paths in the graphs as proxies for deriving new constraints that apply to our original problem. In this way, we can leverage highly efficient graph algorithms as a way to better explore the constraint space and understand whether we can construct a schedule for our STNU. For any path through the graph, we can derive a new implicit constraint based on the endpoints of that path with

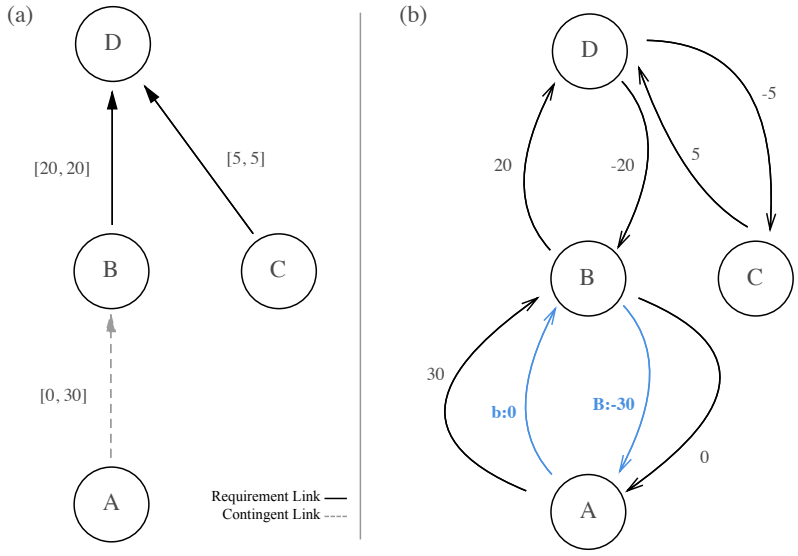

Figure 6: (a) An STNU composed of one contingent link and two requirement links; (b) The same STNU represented as a labeled distance graph. The labeled edges, which represent contingent constraints, are in blue.

weight equal to the weight of the path and whose label is the union of all edge labels in the path.

Because these constraints exist in the temporal domain, we can do even more with them. Our goal is to, whenever possible, make inferences to take a conditional constraint and make it unconditional. This process simplifies our reasoning about the network and makes the problem of determining controllability computationally tractable. Morris and Muscettola (2005) originally introduced a set of edge generation rules, also called reductions, that were used to do exactly this kind of inference for dynamic controllability checking. We can extend and generalize these rules by including modifications necessary to model non-zero delays in observation of received timepoints. The rules are summarized in Table 1 and their application is always with respect to some delay function γ. We now prove that these rules that generate new constraints are sound in that the newly derived edges that are added to the labeled distance graph represent valid constraints on the original STNU.

First, we consider the no-case and upper-case rules. These rules describe how to combine constraints in the typical fashion to produce new constraints, using the telescoping nature of edge-based constraints.

Lemma 4.1. ***No-Case Rule***
If we have $A \xrightarrow{u} B$ *and* $B \xrightarrow{v} C$*, then we can add edge* $A \xrightarrow{u+v} C$*.*

Proof. The first two edges imply that $x_B - x_A \leq u$ and $x_C - x_B \leq v$. Summing the two constraints, we get that $x_C - x_A \leq u + v$. □

Edge Generation Rules			
	Input edges	Conditions	Output edge
No-Case Rule	$A \xrightarrow{u} B$, $B \xrightarrow{v} C$	N/A	$A \xrightarrow{u+v} C$
Upper-Case Rule	$A \xrightarrow{u} D$, $D \xrightarrow{C:v} B$	N/A	$A \xrightarrow{C:u+v} B$
Lower-Case Rule	$A \xrightarrow{c:x} C$, $C \xrightarrow{w} D$	$w < \gamma(C)$, $C \neq D$	$A \xrightarrow{x+w} D$
Cross-Case Rule	$A \xrightarrow{c:x} C$, $C \xrightarrow{B:w} D$	$w < \gamma(C)$, $B \neq C \neq D$	$A \xrightarrow{B:x+w} D$
Label Removal Rule	$B \xrightarrow{C:u} A$, $A \xRightarrow{[x,y]} C$	$u > -x$	$B \xrightarrow{u} A$

Table 1: Edge generation rules for a labeled distance graph

Lemma 4.2. ***Upper-Case Rule***
If we have $A \xrightarrow{u} D$ and $D \xrightarrow{C:v} B$, then we can add edge $A \xrightarrow{C:u+v} B$.

Proof. The first edge gives us the constraint $x_D - x_A \leq u$. The second provides a conditional constraint that $x_B - x_D \leq v$ if C were to take on its maximum value. Summing the constraints, we get a conditional constraint implying we must satisfy $x_B - x_A \leq u + v$ if C were to take on its maximum value. □

Next come the lower-case and cross-case rules. These are the first rules that take advantage of the temporal nature of STNUs. In certain instances we have to assign values to activated timepoints before we can observe the outcome of some uncontrollable action taht they are related to. In those instances, we have to assume the worst-case outcome of the contingent links.

Lemma 4.3. ***Lower-Case Rule***
If we have $A \xrightarrow{c:x} C$, $C \xrightarrow{w} D$, $C \neq D$, and $w < \gamma(C)$, then we can add edge $A \xrightarrow{x+w} D$.

Proof. Since $w < \gamma(C)$ (and $C \neq D$), D must occur before we observe the value of C. For the sake of contradiction, assume that $A \xrightarrow{x+w} D$ was not a valid constraint and that it is possible to assign x_D to occur at some time $t > x + w$ after x_A occurs. If we later learn that x_C happened exactly x units of time after A then we have that $D - C > w$, which violates the original constraint, $C \xrightarrow{w} D$. Thus, we must enforce the constraint $x_D - x_A \leq x + w$, yielding the edge $A \xrightarrow{x+w} D$ in our labeled distance graph. □

Lemma 4.4. ***Cross-Case Rule***
If we have $A \xrightarrow{c:x} C$, $C \xrightarrow{B:w} D$, $B \neq C$, $C \neq D$, and $w < \gamma(C)$, then we can add edge $A \xrightarrow{B:x+w} D$.

Proof. The reasoning for this is the same as the one for the lower-case reduction. If B were to take on its maximum value, we have to plan to satisfy $x_D - x_C \leq w$. In that case, we would have to assign x_D before we observe the value of C. That means we have to be prepared to execute D less than $x + w$ time after A, under the condition that B is to take on its maximum value. □

We refer to these first four edge generation rules as binary rules since they require two edges in the labeled distance graph to produce a third. The label removal rule modifies a

single edge based on the structure of the original STNU and similarly takes advantage of the fact that we may have to enforce a conditional constraint before learning whether the constraint's condition applies.

Lemma 4.5. ***Label Removal Rule***
If we have $B \xrightarrow{C:u} A$, $A \xRightarrow{[x,y]} C$, $u > -x$, *then we can add* $B \xrightarrow{u} A$.

Proof. In this scenario, whenever C takes on its maximum value, we know that A happens at most u after B or, in other words, that B happens at least $-u$ after A. Since $-u < x$, we know that we will not learn about C's actual value before we assign values that satisfy this constraint. As such, instead of having to satisfy $B \xrightarrow{C:u} A$ whenever C takes on its maximum value, we have to satisfy it unconditionally. This is the same as saying that we always have to satisfy $x_B - x_A \geq -u$ or that $B \xrightarrow{u} A$ is a valid edge in our distance graph. □

While this set of rules does not encompass all possible edge generation rules that we could derive, we will show that this choice of rules is sufficient to determine whether an STNU is delay controllable.

In the case of STNs, the presence of a negative cycle indicates that the temporal network is inconsistent. Unfortunately the presence of a negative cycle in an STNU's labeled distance graph does not imply that the STNU is uncontrollable. Because some of the edges only enforce constraints conditionally, a randomly discovered negative cycle may rely on two conditions that are never simultaneously true. In order to determine whether an STNU is delay controllable with respect to γ, we extend the concept of a *semi-reducible negative cycle* from the dynamic controllability literature (Morris, 2006).

Definition 10. Semi-Reducible Negative Cycle (Morris, 2006)
A semi-reducible negative cycle through a labeled distance graph is a cycle that, after a series of reductions with respect to delay function γ, can be transformed into a negative-weight cycle that has no lower-case edges. Since the edge generation rules are parameterized in terms of a delay function, it is important to note that a semi-reducible negative cycle exists with respect to a particular delay function γ.

5. Finding Semi-Reducible Negative Cycles

As in the case for dynamic controllability, the presence of a semi-reducible negative cycle means that an STNU is not delay controllable with respect to γ. Before we prove that these two concepts are equivalent, we provide an algorithm for detecting semi-reducible negative cycles in an STNU, which is based off of an $O(n^3)$ algorithm for determing dynamic controllability in STNUs (Morris, 2014).

5.1 Algorithm Operation

To simplify the execution of our algorithm, we will assume that our STNU is in *normal form*. To transform an STNU into normal form, we eliminate all contingent links $A \xRightarrow{[x,y]} C$ and introduce a new timepoint A' with two links, $A \xrightarrow{[x,x]} A'$ and $A' \xRightarrow{[0,y-x]} C$. We apply

this transformation even if $x = 0$. From an expressiveness perspective, the two STNUs are equally expressive, meaning the transformation does not affect delay controllability.

Input: A labeled distance graph, $G = \langle V, E \rangle$; A delay function γ
Output: Whether the STNU derived from the distance graph is delay controllable with respect to γ
Initialization:
1 $negNodes \leftarrow$ the set of all vertices with incoming negative edges;
DelayControllable?:
2 **for** $v \in negNodes$ **do**
3 | $cycleFree? \leftarrow$DelayDijkstra(G, γ, v, $[v]$, $negNodes$);
4 | **if** !$cycleFree?$ **then**
5 | | **return** $false$;
6 **return** $true$;

Algorithm 1: Delay Controllability algorithm

To provide an intuition how Algorithm 1 finds semi-reducible negative cycles, it is easier to first look at its main subroutine DelayDijkstra (Algorithm 2). DelayDijkstra attempts to find the shortest semi-reducible path from all nodes in the graph to terminal node s. However, because Dijkstra's algorithm is generally incapable of handling negative edges, our algorithm ignores all negative edges except those that start our shortest path walk.

DelayDijkstra works by walking along negative paths, hoping that those paths eventually become positive. The decision to walk edges in reverse from terminal node s to all other nodes is used to simplify the guarantee that walks are along semi-reducible paths. A lower-case edge with label c can only be reduced if followed by an edge with weight less than $\gamma(x_c)$. It is difficult to look-ahead and see whether some set of reductions could produce an edge with low enough weight, but when walking in reverse it is easy to see that everything that has been walked so far has a weight that comes under the threshold. If a walk along a semi-reducible path eventually takes on a non-negative weight, the algorithm reduces the path down to a single edge, adds it to the graph, and continues its walk.

So far our sketch has ignored most negative edges. It may be that we need to traverse some of those negative edges for our walk to eventually become non-negative or to discover a semi-reducible negative cycle. Whenever the algorithm reaches a node that has an incoming edge with negative weights, it recursively calls DelayDijkstra on that node. The recursive call either completes successfully, in which case all of the non-negative extensions of that edge have been added to the graph, or it continues recursively calling DelayDijkstra as its walk continues. If we infinitely recurse, we know we have found a semi-reducible negative cycle since each component semi-reducible path is negative (line 8 of Algorithm 2).

Algorithm 1 serves as a wrapper function ensuring that DelayDijkstra is called on all nodes. If one of the subroutine calls finds a semi-reducible negative cycle, we return false. If none of them find a semi-reducible negative cycle, then we return true.

To illustrate how the algorithm works, we give a brief sketch of how we perform delay controllability checking on the example in Figure 7, which contains a subset of the constraints from Example 4 that together still render the network uncontrollable.

Input: Labeled distance graph $G = \langle V, E \rangle$, delay function γ, terminal node s, $callStack$, and negative nodes $negNodes$
Output: Whether the current walk is cycle-free
Initialization:

```
1  Q ← PriorityQueue();
2  distances ← [];      # shortest distances for semi-reducible path;
3  distances[⟨s, ∅⟩] ← ⟨0, ∅⟩;
4  for e ∈ s.incomingEdges() do
5  |  if e.weight < 0 and !e.lowerCase() then
6  |  |  Q.add(⟨e.from, e.label⟩, e.weight);
7  |  |  distances[⟨e.from, e.label⟩] ← ⟨e.weight, e⟩
```

DelayDijkstra:

```
8  if s ∈ callStack[1 : end] then
9  |  return false;
10 while Q.size() > 0 do
11 |  v, label, weight ← Q.pop();
12 |  if weight ≥ 0 then
13 |  |  G.add(⟨v, s, weight⟩);
14 |  else
15 |  |  if v ∈ negNodes then
16 |  |  |  newStack ← [v].concat(callStack);
17 |  |  |  result ← DelayDijkstra(G, γ, v, newStack, negNodes);
18 |  |  |  if !result then
19 |  |  |  |  return false;
20 |  |  for e ∈ v.incomingEdges() do
21 |  |  |  if e.weight ≥ 0 and (!e.isLowerCase() or e.label ≠ label) then
22 |  |  |  |  w ← e.weight + weight;
23 |  |  |  |  if Q.addOrDecKey(⟨e.from, label⟩, w) then
24 |  |  |  |  |  distances[⟨e.from, label⟩] ← ⟨w, e⟩;
25 |  |  |  |  lower ← (e.from).incomingLowerEdge();
26 |  |  |  |  if lower ≠ null and e.weight < γ(lower.label) then
27 |  |  |  |  |  if Q.addOrDecKey(⟨lower.from, label⟩, w + lower.weight) then
28 |  |  |  |  |  |  distances[⟨lower.from, label⟩] ← ⟨w + lower.weight, lower⟩;
29 negNodes.remove(s);
30 return true;
```

Algorithm 2: Function DelayDijkstra

We start by collecting all nodes that have incoming negative edges in the labeled distance graph, which gives us $[A, B, D]$, and start performing DelayDijkstra on the nodes of the list. We start with A, enqueuing all incoming edges with negative weight, and then start our walk using Dijkstra's algorithm. The first node we dequeue is B, which is reached using $B \xrightarrow{B:-40} A$, and because B has incoming negative edges we recurse.

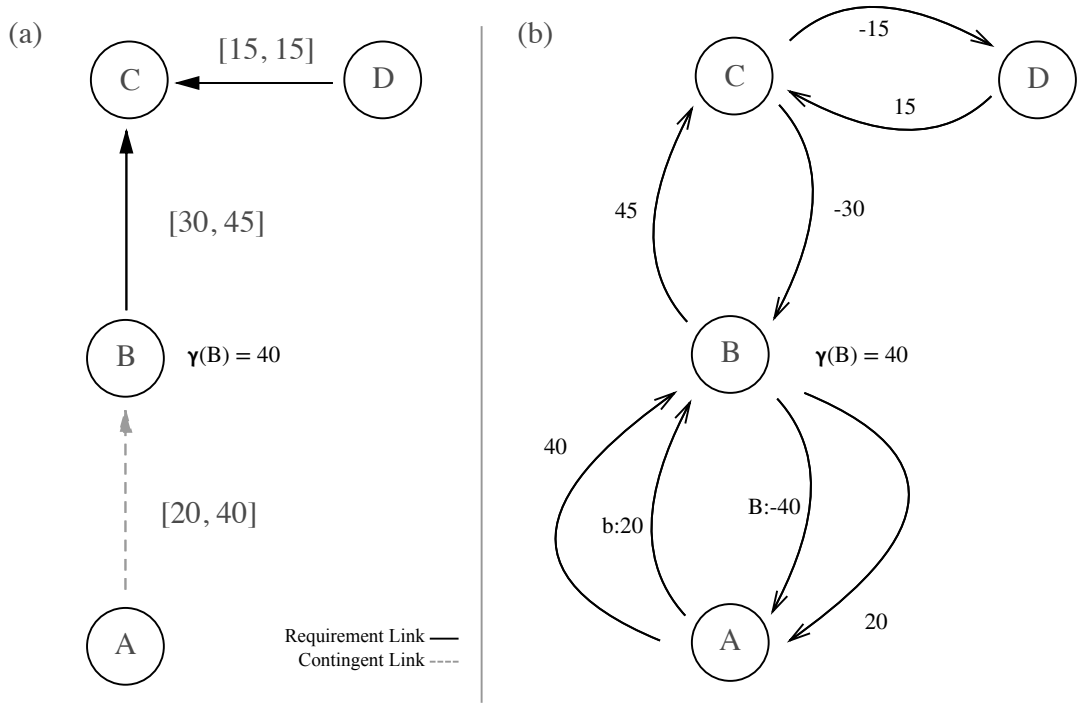

Figure 7: (a) Example of an STNU that is uncontrollable when $\gamma(B) = 40$. (b) The same STNU represented as a labeled distance graph.

We repeat the process with B and next dequeue the node C adding its successors. Next to be dequeued is D which is of distance -15 away from B. Because D is also a negative node, we recurse again.

With our recursive call to D, our algorithm terminates for the first time, as we enqueue C using $C \xrightarrow{-15} D$, and when we dequeue C, the only edge that leads us to a new, shorter path is $B \xrightarrow{45} C$. When we reach B, DelayDijkstra terminates and adds $B \xrightarrow{30} D$ to the graph as a newly derived semi-reducible path. Since it terminates, we return to the recursive call that had initial terminal node B.

In our previous recursive call, we had just reached D and so now want to consider its non-negative extensions. One of the edges we would consider is the newly added $B \xrightarrow{30} D$. However, we would not enqueue this extension, as the path $C \xrightarrow{-30} B, D \xrightarrow{15} C, B \xrightarrow{30} D$ does not yield a shorter path and so fails the check at line 23 of Algorithm 1. However, we still must perform the lookahead at lines 25-28. Since the weight of the new $B \xrightarrow{30} D$ is less than $\gamma(B)$, we implicitly apply the lower-case reduction to create $A \xrightarrow{50} D$ from $A \xrightarrow{b:20} B$ and $B \xrightarrow{30} D$. When our continuation of Dijkstra's algorithm eventually reaches A, we add the new edge $A \xrightarrow{35} B$ and return.

We finally return to our original recursive call that had an initial terminal node of A. We were about to consider B's extensions and now have the new edge $A \xrightarrow{35} B$. But by taking that edge, we will return back to A since our path of $B \xrightarrow{B:-40} A$ and $A \xrightarrow{35} B$ has total weight -5. A has incoming negative edges, meaning we recurse again. However, because A was already in our call stack (line 8), we return false, meaning the STNU is not delay controllable.

5.2 Correctness

We now move on to rigorously prove that Algorithm 1 returns true if and only if there are no semi-reducible negative cycles.

Lemma 5.1. *If Algorithm 1 returns false on STNU S and delay function γ, then S has a semi-reducible negative cycle with respect to γ.*

Proof. Algorithm 1 only returns false if a recursive call to DelayDijkstra returns false, which only happens if terminal node s was already present in the call stack somewhere. Consider the situation that would have s show up in the call stack twice.

We know that we only make a recursive call if the weight of the path we are considering is negative (lines 14, 17). This means that if s shows up in the call stack twice, we have a series of negative paths that start and end at s, implying that we have found a negative cycle.

In order to prove that it is a semi-reducible negative cycle, we show that each path popped from Q is a semi-reducible path. First, we note that for a particular terminal node s, there is at most one label that can be assigned to any path that we build with DelayDijkstra. Because our STNU is in normal form, no upper-case labeled edges share an endpoint, and we also know that no upper-case labeled edges share their endpoint with an unlabeled negative edge. This means that at line 6, we only consider unlabeled edges and at most one upper-case edge. When we continue through execution, we know our path will never adopt another label value. All upper-case edges are initially negative and so are ignored by line 21. However, we do have to consider upper-case edges that are positive and added to the graph by a different call at line 13. But because the label removal rule gives us a guarantee that we can always apply it to an upper-case edge with non-negative weight, we can elect to only add unlabeled edges to the graph.

Since we know that for a particular call to DelayDijkstra that we only have to consider one type of label, ensuring that our path is semi-reducible becomes much simpler. At line 21, we ensure that we do not do an illegal cross-case reduction due to labels matching, and at line 26, we make sure that we only apply cross-case or lower-case reductions if the existing path weight is within the bounds specified by γ. We do not have to perform the same type of check at line 21 because the else at line 14 guarantees that the total weight of the path so far is negative. Thus, the paths that we build up are semi-reducible, and if our main algorithm returns false, we have a guarantee that the STNU has a semi-reducible negative cycle under γ. □

To complete the proof, we must also show that if an STNU has a semi-reducible negative cycle, then our algorithm will find one.

Lemma 5.2. *If S has a semi-reducible negative cycle with respect to delay function γ, then Algorithm 1 returns false.*

Proof. Consider any semi-reducible negative cycle that contains no negative semi-reducible sub-cycles. We know that we can partition the negative cycle into a series of semi-reducible paths such that every path has only one negative weight edge and that edge is the final edge in the path. Note that some of these partitions may consist of just a single negative node.

We know that there must be at least one negative edge such that if we walked backwards along the semi-reducible negative cycle starting from that edge, our weight never becomes non-negative. If not, we could take every such eventually non-negative path extension and remove those edges from the cycle. The only edges remaining would be non-negative ones, and we would arrive at a contradiction since our original cycle had negative total weight.

Consider a negative edge where when we walk backwards from that edge along the semi-reducible path, the path weight becomes non-negative. We know that if DelayDijkstra is called on that edge, that path will be collapsed and added to the graph as a single edge (line 13), as the *addOrDecKey* method only modifies our queue if there are shorter weights which guarantees that we will find the path with shortest weight. In our semi-reducible negative cycle, we will replace that path with the newly generated edge. We can repeat this process until we have no more edges whose walks eventually become non-negative.

Since we are interested in eliminating lower-case edges, we need to make sure that every lower-case edge that is used in a semi-reducible negative cycle is found and reduced away. Since our algorithm starts at a negative edge and then traverses non-negative edges forward until the semi-reducible path weight becomes non-negative, we know that whenever our weight is yet to become positive, we can always take a lower-case edge. However, in instances where $\gamma > 0$, it is possible that we wanted to continue our walk to eliminate a lower-case edge instead of terminating at the first non-negative edge. This is especially true when we have a path of the form $A \xrightarrow{b:x} B \rightsquigarrow B$, and we want to eliminate the lower-case edge with some subsequence of the path instead of the entire path. Handling this look-ahead reduction appropriately is one of the main complications of the introduction of delay functions.

Because we always start with a negative edge and add only non-negative edges, the two easiest ways to reduce a lower-case edge are by reducing it against the whole path, or by replacing it with its immediate predecessor. Any longer subsequence of edges that does not include the initial negative edge is going to have cost at least as large, and we can only reduce the lower-case edge if the total path length is less than γ. Lines 26-28 guarantee that we always do a lookahead for any incoming lower-case edges to see if they can be immediately reduced against the current edge that we are considering. If they can, we immediately perform the reduction and enqueue the combined edge. If not, we do nothing and let the edge be handled normally.

For the negative edges that remain, we know that if we were to run DelayDijkstra from any one of them, our algorithm would return false since the successful completion of any one DelayDijkstra call is dependent on the next. The algorithm would continue walking the shortest semi-reducible paths making recursive calls until the call stack detected that there was infinite recursion (line 8). Since our main algorithm guarantees that we will

eventually call DelayDijkstra for all nodes with negative incoming edges, we have a guarantee that if there is a semi-reducible negative cycle, we will return false. □

Stringing the pieces together, we can now show that our algorithm is sound and complete and also that it runs efficiently.

Theorem 5.3. *Algorithm 1 is a sound and complete algorithm for determining whether an STNU is free of semi-reducible negative cycles with respect to delay function γ in $O(n^3)$ time.*

Proof. By Lemmas 5.1 and 5.2, we know that Algorithm 1 is sound and complete. What remains is to show is that it completes in $O(n^3)$ time.

First note that we only call DelayDijkstra for a terminal node s if we first verify that $s \in negNodes$ (lines 2-3 of Algorithm 1; line 15 of Algorithm 2). Whenever DelayDijkstra returns true for terminal node s, we know it has been removed from $negNodes$ (line 29 of Algorithm 2), and whenever DelayDijkstra returns false, we immediately return false for all ancestor calls (lines 3-5 of Algorithm 1 and lines 17-19 of Algorithm 2). Because we never add values to $negNodes$, this means that we call DelayDijkstra at most n times in total.

Now we analyze the runtime of DelayDijkstra independent of any recursive calls. We are implicitly running two versions of Dijkstra's algorithm simultaneously, one with labeled paths and one with unlabeled paths. While this doubles the number of items added to the queue and the number of overall operations, the doubling has no effect on the overall runtime of Dijkstra's algorithm, which is $O(m + n \log n)$.

However, when we consider the total number of edges, it is more than the initial m that belong to the labeled distance graph G. Each time DelayDijkstra is called, n new edges are potentially added to the graph (line 13), meaning to be safe we must assume that there are $O(n^2)$ edges in total, raising the total runtime of DelayDijktsra to $O(n^2)$.

Putting it all together, we run DelayDijkstra at most n times, giving us a total worst-case runtime of $O(n^3)$. □

6. Verifying Delay Controllability

We now have an efficient algorithm that detects whether or not there are semi-reducible negative cycles in an STNU's labeled distance graph. However, our original goal was to determine whether an STNU was delay controllable with respect to some delay function γ. To do so, we need to show that an STNU is delay controllable if and only if its labeled distance graph has no semi-reducible negative cycles.

Lemma 6.1. *If the labeled distance graph of some STNU, S, has a semi-reducible negative cycle with respect to delay function γ, then S is not delay controllable with respect to γ.*

Proof. If we have a semi-reducible negative cycle, we know that we can apply a series of reductions to those edges to yield a negative cycle of only upper-case and unlabeled edges. We can combine all unlabeled edges together using the no-case reduction or into upper-case edges using the upper-case reduction and then transform upper-case edges into unlabeled ones if label removal can be applied. Since the upper-case reductions reduce the number of

present edges, we can iteratively apply this until we are either left with a single unlabeled self-edge or a cycle of upper-cases edges that cannot undergo a label reduction.

If we end with a single self-edge, then any projection of contingent durations will yield an inconsistent STN since the negative self-edge is a negative cycle. In this case, we know that S is not delay controllable with respect to γ.

If we end with a cycle of upper-case edges, then each edge is of the form $B \xrightarrow{C:u} A$ where the original contingent link was of the form $A \xRightarrow{[x,y]} C$ and $u < -x$. This means that if we have not yet inferred that C has occurred, the condition $x_a - x_b \leq u < -x$ holds. Since $x \geq 0$, we have that $x_a - x_b < 0$. However, we get this equation for all edges in our cycle. Because we are evaluating delay controllability prior to execution, we cannot infer that any contingent links have yet occurred, and so can evaluate all of these constraints simultaneously. Summing them, we get that $0 < 0$, yielding a contradiction.

Thus, whenever we find a semi-reducible negative cycle with respect to γ for STNU S, we know that S is not delay controllable with respect to γ. □

In order to show that an STNU free of semi-reducible negative cycle is delay controllable, we will show that in those situations we can always come up with an execution strategy that satisfies all constraints and only observes the outcomes of contingent links after their specified delay has elapsed. Our process for generating an execution strategy will closely follow the process of doing so for dynamic controllability as demonstrated in Hunsberger (2016), as will its proof of correctness. Before we specify an execution strategy, we need to prove a few properties of STNUs that are free of semi-reducible negative cycles to show that the notion of a shortest semi-reducible path is well-defined. First, we show that the number of possible edges we can derive is bounded.

Lemma 6.2. *If an STNU does not have a semi-reducible negative cycle, then a finite number of edges can be generated with the edge generation rules. In particular, no more than $O(n^3)$ rounds of applications of edge generation rules are needed to generate all possible edges.*

Proof. We define the following edge generation strategy. Let E_0 be the set of edges we start with in our labeled distance graph. To generate the set of edges E_{i+1} from E_i, we take all pairs of edges from E_i and see if we can apply any of the binary edge generation rules to produce a new edge. Then for each new edge we generate, we try to apply the Label Removal rule to see if we can generate additional edges. We take all edges from E_i and all newly generated edges and for each triple (start node, end node, label), we store the shortest edge for that triple in E_{i+1}. We terminate when the set of output edges remains unchanged after a round of generation.

If there are no semi-reducible negative cycles, then there will be at most $n^3 + n^2$ rounds of edge generation. We know that for any edge e that newly shows up in round $i + 1$, at least one of its parent edges was generated in round i. If not, e would have been generated in an earlier round and in round $i + 1$, it would be discarded since it was not the shortest edge for its corresponding triple. Thus, if there are k rounds, we can take an edge generated in round k and back out a sequence of edge transformations responsible for creating it and know that we have at least one edge from the sequence generated at each round. We always pick the shortest edge for any particular start node, end node, and label triple during edge generation, meaning there are at most $n^3 + n^2$ such triples if we ignore lower-case edges

since there are n^2 start and end node pairs and $n+1$ possible labels, including the unlabeled state. It is safe for us to ignore lower-case edges in the output of edge generation since none of the edge generation rules produce lower-case edges.

Assume for the sake of contradiction that there are $k > n^3+n^2$ rounds of edge generation. We can take an edge generated in round k and back out a series of k edge transformations, one per round, that were necessary to generate that edge. Of the k outputs, we know that at least two edges must share the same triple (start node, end node, label). Call the rounds that outputted those two edges i and j and the corresponding edges e_i and e_j. If $i < j$, we also know that the weight of the edge outputted in round j is less than the weight of the edge outputted in round i. For convenience, we call the start node of our edges A and their end node B.

We will show that if there is a series of transformations that we can apply to e_i to produce e_j, then there must exist a semi-reducible negative cycle and we have a contradiction. Since the iterative application of edge generation rules either extends an edge forwards or backwards, we can model the transformation from e_i to e_j as a path involving e_i and several other edges where the final path has the same start and endpoint as e_j. Our path has $j-i+1$ edges so after exactly $j-i$ applications of the edge generation rules, we would create e_j.

If we consider this path, we notice that it contains subpaths from the start of e_j (A) to the start of e_i (A) and from the end of e_i (B) to the end of e_j (B). Both subpaths are cycles (with one possibly being empty), and since the weight of e_j is less than the weight of e_i, we know that at least one of the cycles is negative. Now we show that if one of these cycles is negative, it is also semi-reducible.

First consider the case where the path $B \rightsquigarrow B$ has negative weight. We know that these edges are applied one at a time in order to some intermediate products such that we eventually get e_j. This means that each one of these edges acts as the successor in one of the four binary edge generation rules. We notice that the only valid edges that can come in that position are upper-case and unlabeled edges, so by definition this cycle is semi-reducible.

Now assume that the path $B \rightsquigarrow B$ is non-negative. This implies that the $A \rightsquigarrow A$ has negative weight. Here again, we know that these edges are applied one at a time to an intermediate product, but instead, these edges act as predecessor edges in the binary edge generation rules. This means that they must all be either lower-case or unlabeled edges.

We know that a negative cycle of all lower-case or unlabeled edges is semi-reducible. Since all lower-case edges have non-negative weight, the sum of the weights of all unlabeled edges in the cycle must be strictly less than zero. If a lower-case edge exists in the cycle, there must be at least one such edge that is followed by a series of unlabeled edges whose combined weight is negative since all lower-case edges have non-negative weight. Since $\forall x_e \in X_e, \gamma(x_e) \geq 0$, we know we can apply a series of no-case reductions followed by a lower-case reduction to eliminate one lower-case edge. These invariants continue to hold for any negative cycle composed solely of lower-case or unlabeled edges, so we can continue eliminating lower-case edges in this way until they are all gone. This shows that our subcycle is semi-reducible.

If edge generation is unbounded, we will have a series of transformations that transform e_i into e_j and will be able to find a semi-reducible negative cycle. Thus, if there are no

semi-reducible negative cycles, then the edge generation rules can only produce a finite number of edges, all of which can be found after $O(n^3)$ rounds.

□

It is important to note that our approach for deriving all possible edges is different than the approach of the DelayDijkstra algorithm, as that algorithm does not rely on generating all possible edges. However, knowing that there is a limit to the number of possible edges that can be generated is useful for determing whether the notion of a shortest semi-reducible path is well-defined in an STNU. Since we know that whenever an STNU is free of semi-reducible negative cycles there are a finite number of edges we can derive in that STNU's labeled distance graph, we can show that the notion of a shortest semi-reducible path is well-defined and, importantly, computable.

Lemma 6.3. *If an STNU does not have a semi-reducible negative cycle, then for every pair of nodes A, B in the STNU's labeled distance graphs, there exists some k that is less than the shortest semi-reducible path between A and B, meaning that the notion of a shortest semi-reducible path is well defined.*

Proof. Assume that our STNU does not have a semi-reducible negative cycle and consider the labeled distance graph produced after generating all possible edges. By Lemma 6.2, we know that there are a finite number of generated edges.

Examine any semi-reducible path between A and B. We know that since all possible edges are generated and included in our labeled distance graph, we can construct an equivalent semi-reducible path from our set of all edges that has the same weight but has no lower-case edges. If our path has any sub-cycles, we know we can remove them without increasing the path length since our STNU is free of semi-reducible negative cycles and all of our sub-cycles are free of lower-case edges and thus semi-reducible. But if we look at the resulting semi-reducible path, it is bounded in weight (since there are only finitely many possible edges to pick from). Thus, if an STNU is free of semi-reducible negative cycles, then there is a shortest semi-reducible path between all nodes.

□

Because shortest semi-reducible paths are well-defined, we can incorporate their use into an execution strategy for STNUs. Whenever an STNU has a valid exeuction strategy, we know that the STNU is delay controllable, so if we can construct an execution strategy for any graph on which semi-reducible shortest paths is well-defined, we know that not having a semi-reducible negative cycle is sufficient for being delay controllable. Much of this proof resembles the one from (Hunsberger, 2016), but special care is taken to show that the introduction of delays does not introduce new problems in the main arguments of the proof.

Lemma 6.4. *If the labeled distance graph of some STNU, S, does not have a semi-reducible negative cycle with respect to delay function γ, then S is delay controllable with respect to γ.*

Proof. We will prove this claim by specifying an execution strategy that is valid as long as the shortest semi-reducible path is well-defined on a labeled distance graph. Then we show

that our execution strategy always maintains the invariants that the labeled distance graph is free of semi-reducible negative cycles and that we never exceed an activated timepoint's upper bound for possible execution.

We derive the lower and upper bounds for a timepoint's execution by looking at the shortest semi-reducible path between Z and other nodes, where Z represents the earliest occurring activated timepoint which by convention occurs at 0. If the shortest semi-reducible path between Z and B is u while the shortest semi-reducible path between B and Z is $-l$, we say that B has lower bound l and upper bound u. By Lemma 6.3, we know that the shortest semi-reducible path is well-defined, but for this proof we do not need an efficient means of computing it.

At any given time t, we schedule our next timepoint to execute as follows. First, we search across all activated timepoints which have yet to be assigned and find the ones with the earliest lower bounds. If the selected timepoints have lower bound greater than t, our plan is to execute them at their lower bound. Otherwise, we plan to execute that timepoint immediately (at time t), as our model allows for instantaneous execution. Every time we activate a timepoint or observe a received timepoint, we re-determine what the next timepoint to execute is. After all executed timepoints have been specified, we reveal the remaining unobserved received timepoints and verify that the resulting STNU projection is consistent.

Along the way, we also make sure to remove any constraints that are no longer valid. In particular, for any contingent link $A \xRightarrow{[x,y]} C$, if $\gamma(C) + x$ time has passed since A was executed, we can safely remove the lower case edge $A \xrightarrow{c:x} C$. That lower-case edge represents a bound that must be enforced if the contingent link were to take on its lowest possible value, but because $\gamma(C) + x$ time has passed, we know that it cannot have its minimum possible value.

We now proceed with some case analysis to show that our execution strategy is sound. First we consider what happens when we observe a received timepoint. Then we consider what happens when we execute an activated timepoint. Whenever we know the new value of a timepoint B (whether we assigned it or received it) that occurred at time t, we fix that point in our labeled distance graph by adding links $Z \xrightarrow{t} B$ and $B \xrightarrow{-t} Z$. We will show that in either case, if the original graph was free of semi-reducible negative cycles, then the resulting graph will also be free of semi-reducible negative cycles. We will also show that at every step along the way, no unexecuted timepoint's upper bound ever drops below the current time.

Case 1 - Received Timepoint. Assume that we just observed the value of $A \xRightarrow{[x,y]} C$ and see that C occurred at time t. Since $x, y \geq 0$ and $\gamma(C) \geq 0$, we know that activated timepoint A must have been scheduled before we observed the value of C. Say that A occurred at time τ. We can now add the links between Z and C of weight t and $-t$. In addition to adding the two new links, we should remove the $A \xrightarrow{c:x} C$ link and the $C \xrightarrow{C:-y} A$ link. Those links represented the fact that $A \to C$ could possibly take on any value between x and y, but because we know the true value of that link, it is incorrect to make inferences based on values that we know it cannot take on. For the sake of contradiction, assume that the addition of our two new links created a semi-reducible negative cycle.

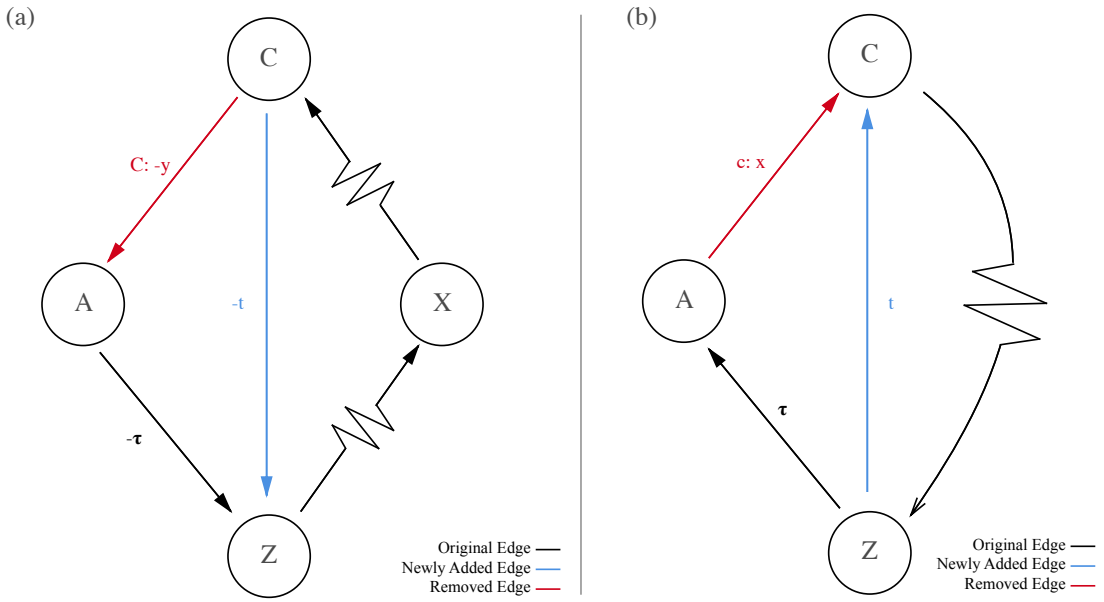

Figure 8: (a) Diagram indicating that the observation of C occurring at time t creates no semi-reducible negative cycles that use $C \xrightarrow{-t} Z$. The path $X \rightsquigarrow C$ starts with a lower-case edge if it is non-empty. (b) Another diagram indicating the same thing for the other added edge, $Z \xrightarrow{t} C$.

We know that $C \xrightarrow{-t} Z$ cannot be part of a semi-reducible negative cycle. If it were, then after modifying the set of edges in the graph we would have some semi-reducible path $Z \rightsquigarrow X$ and some (possibly empty) path $X \rightsquigarrow C$ starting with a lower case edge, such that the total weight of their paths is α, where $\alpha < t$ (see Figure 8a).

If $X \rightsquigarrow C$ is empty, we know that $Z \rightsquigarrow C$ is a semi-reducible path, and a contradiction immediately follows. $Z \rightsquigarrow C$ does not have a lower-case c edge since that edge had already been removed. This means that we can perform an upper-case reduction with $C \xrightarrow{C:-y} A$ and $Z \rightsquigarrow C$ and then combining it with $A \xrightarrow{-\tau} Z$, we have a semi-reducible cycle with weight $\alpha - y - \tau < t - y - \tau$. But we also know that $t \leq \tau + y$ by construction, so we have that $\alpha - y - \tau < 0$ and thus we would have already had a semi-reducible negative cycle, which is a contradiction.

Now we consider what happens if $X \rightsquigarrow C$ is not empty and the newly added edge provides us a negative cycle. This means that the addition of an edge with weight $-t$ would be enough to reduce the lower-case edge. But by the same reasoning, we know that $X \rightsquigarrow C$ does not have a lower-case c edge, and we could have instead substituted in the combination of the more negative $C \xrightarrow{C:-y} A$ and $A \xrightarrow{-\tau} Z$ to achieve the same reduction, and we would have violated our precondition that we start with no semi-reducible negative cycles. Thus, the new edge $C \xrightarrow{-t} Z$ cannot be part of a semi-reducible negative cycle.

We also know that $Z \xrightarrow{t} C$ cannot be part of a semi-reducible negative cycle. If it were, there would be some semi-reducible path $C \rightsquigarrow Z$ with weight α, where $-\alpha > t \geq 0$ (see Figure 8b). In that case, before we substituted in the edges, we could have created a semi-reducible negative cycle with the semi-reducible path $C \rightsquigarrow Z$, $Z \xrightarrow{\tau} A$, and $A \xrightarrow{c:x} C$. In order for this to be the case, there must have been a semi-reducible path $A \rightsquigarrow Z$ that uses $A \xrightarrow{c:x} C$ and $C \rightsquigarrow Z$. We know that the semi-reducible path $C \rightsquigarrow Z$ with weight α does not have any upper-case C edges because we removed them. The only way then that we would be unable to apply the lower-case rule using $A \xrightarrow{c:x} C$ and $C \rightsquigarrow Z$ would be if C were immediately followed by some series of edges that collapsed down to either $C \xrightarrow{D:\beta} B$ or $C \xrightarrow{\beta} B$, where $\beta > \gamma(C)$. In the case where the reduced edge has label D, we can apply the label removal rule since $\beta > \gamma(C) \geq 0 \geq -x_D$, meaning we only have to consider the case where C is followed by $C \xrightarrow{\beta} B$.

Since $\alpha < 0$, we know that there must be more edges in the sequence, and we can continue reducing forward using the no-case or upper-case rules. At every point, if the weight of the path is greater than β, we can apply the label removal rule and continue reducing forward. Eventually, the value of the edge will drop below β since the total weight of the path is $\alpha < 0 < \beta$, meaning that we can apply the lower-case rule.

The semi-reducible cycle composed of $C \rightsquigarrow Z$, $Z \xrightarrow{\tau} A$, and $A \xrightarrow{c:x} C$ has weight $\alpha+\tau+x$. Since $\tau + x \leq t$, our semi-reducible cycle has weight $\alpha + \tau + x \leq \alpha + t < 0$. Thus, we would have had a semi-reducible negative cycle and so whenever we assign a received timepoint, we do not create any new semi-reducible negative cycles.

We also know that no unexecuted timepoint's upper bound will ever dip below the current time, $t+\gamma(C)$. Assume for the sake of contradiction that the upper bound of B was greater than or equal to $t + \gamma(C)$ before we observed the value of C and then dipped to a value of less than $t+\gamma(C)$ afterwards. This can only happen if there is a new semi-reducible path from Z to B with value less than $t + \gamma(C)$. Such a path would have to include one of the new edges. However, it will not include $C \xrightarrow{-t} Z$ because then the full path would be composed of $Z \rightsquigarrow C$, $C \xrightarrow{-t} Z$, and $Z \rightsquigarrow B$. Since there are no semi-reducible negative cycles, we can just look at the shorter $Z \rightsquigarrow B$ directly.

Assume that after adding the new edges, we have a semi-reducible path from Z to B with value less than $t + \gamma(C)$. It must be the case that there is some semi-reducible path $C \rightsquigarrow B$ with weight $\alpha < \gamma(C)$, as the only way the path could decrease is if it took one of the new edges and the only such new edge that qualifies is $Z \xrightarrow{t} C$. But look what would have happened before we substituted in the new edges. If we take $Z \xrightarrow{\tau} A$, $A \xrightarrow{c:x} C$, and the semi-reducible path $C \rightsquigarrow B$ with weight α, we know we can apply the lower-case reduction combining $A \xrightarrow{c:x} C$ and the semi-reducible path $C \rightsquigarrow B$ since $\alpha < \gamma(C)$. That means that we had a semi-reducible path from Z to B with weight $\tau + x + \alpha \leq t + \alpha < t + \gamma(C)$, which violates our initial assumption that the shortest semi-reducible path from Z to B had weight at least $t+\gamma(C)$. Thus, observing a received timepoint will not cause the upper bound of an unexecuted timepoint to drop below the current time.

Case 2 - Activated Timepoint. Assume that we just assigned a value to activated timepoint A at time t. First, we show that this does not introduce any semi-reducible negative cycles.

Imagine that there was a new semi-reducible negative cycle that used edge $Z \xrightarrow{t} A$. That means that there existed some other semi-reducible path $A \rightsquigarrow Z$ with weight $-\alpha < -t$ that existed before the introduction of new edges. However, that would have implied that A's lower bound α was greater than t, and we would not have executed A at this time.

Now assume that there was a new semi-reducible negative cycle that used edge $A \xrightarrow{-t} Z$. That means that there existed some other semi-reducible path $Z \rightsquigarrow A$ with weight $\alpha < t$ that existed before the introduction of new edges. However, that would have implied that A's upper bound should have been α, which is less than t. Since we maintain the precondition that no unexecuted timepoint can have an upper bound less than the current time, we have a contradiction. Thus, the assignment of an executable timepoint A cannot create a new semi-reducible negative cycle.

Now, we show that the assignment of the value t to A cannot cause some other timepoint B's upper bound to fall below t. Assume that the assignment did cause B's upper bound to drop. This implies that after adding the new edges, we have a semi-reducible path from Z to B with weight less than t, and we know that the path must use $Z \xrightarrow{t} A$. It must use a new edge for the weight to drop, and if we used $A \xrightarrow{-t} Z$, the full semi-reducible path would look like $Z \rightsquigarrow A$, $A \xrightarrow{-t} Z$, and $Z \rightsquigarrow B$. Since there are no semi-reducible negative cycles, we would have at least as short a path by just taking $Z \rightsquigarrow B$ directly.

This would imply that there is some semi-reducible path from A to B with weight $\alpha < 0$. Since B has not been executed yet, we also know that its lower bound is $\beta \geq t$ implying that there exists some semi-reducible path $B \rightsquigarrow Z$ with weight $-\beta$. But this means that we have a semi-reducible cycle $Z \xrightarrow{t} A$, $A \rightsquigarrow B$ with weight α, and $B \rightsquigarrow Z$ with weight $-\beta$. However, this yields a contradiction since $t + \alpha - \beta \leq \alpha < 0$, meaning that this would create a semi-reducible negative cycle, which we know cannot happen.

Thus, whenever we receive or activate a timepoint, we keep our STNU free of semi-reducible negative cycles and preserve the executability of our STNU since no executable timepoint's upper bound dips below the current time. Since we have a valid execution strategy, this means that if an STNU is free of semi-reducible negative cycles, it is delay controllable with respect to γ. □

We can now put all of our work together to show that we have an efficient, sound and complete algorithm for computing delay controllability.

Theorem 6.5. *Delay controllability of an STNU S with respect to γ can be checked in $O(n^3)$ time.*

Proof. With Lemma 6.1 and Lemma 6.4, we know that S is delay controllable if and only if S is free of semi-reducible negative cycles. By Theorem 5.3, we know that we can determine whether a semi-reducible negative cycle exists in $O(n^3)$ time, implying that Algorithm 1 gives us an $O(n^3)$ check for delay controllability. □

7. Experimental Results

The introduction of delay controllability gives us a new level of expressiveness that was otherwise unavailable to those modeling their problems with STNUs. Of particular note is the quality of solutions that we get back when using delay controllability. While strong

controllability checks tend to be too conservative and dynamic controllability checks assume a strong degree of flexibility from the perspective of the main agent, delay controllability occupies an appropriate middle ground, accounting for an actor's limitations without being overly restrictive.

In order to capture the difference in quality between the different approaches, we evaluted the outcomes of different controllability checks on a set of randomly generated STNUs. We selected our parameters because they represent a reasonable tradeoff between simplicity in degenerate cases and sufficient complexity to exhibit interesting behaviors. Each STNU had 10 disconnected contingent links with lower bound 0 and an integer upper bound uniformly chosen between 1 and 4. For the delay controllability checks, the observation delay was also an integer uniformly chosen between 1 and 4. For each pair of contingent link endpoints, we created a requirement link between the two with probability $\frac{1}{40}$. Each requirement link had a lower bound of 0 and an integer upper bound uniformly chosen bretween 1 and 4. These parameters were tuned in order to ensure that there was a good mix of controllable and uncontrollable problems.

We constructed 1000 different STNUs, and determined whether they were dynamically, delay, or strongly controllable. In analyzing the results, we noticed that dynamic and strong controllability have high error rates when used to approximate delay controllability. Strong controllability acts as a more conservative version of delay controllability, registering a false negative rate of 21.3% (Table 2). In contrast, dynamic controllability ignores some of the constraints of the problem and shows a false positive rate of 43.1% (Table 3). From a correctness perspective, delay controllability represents a significant improvement above and beyond existing controllability checks. Because such a wide gap exists in both directions, from a solution-quality perspective, there is no reason to prefer using strong or dynamic controllability to model communication delays in the resolution of temporally uncertain events.

	Delay controllable	Delay uncontrollable
Strongly controllable	162	0
Strongly uncontrollable	44	794

Table 2: Delay vs. strong controllability results.

	Delay controllable	Delay uncontrollable
Dynamically controllable	206	342
Dynamically uncontrollable	0	452

Table 3: Delay vs. dynamic controllability results.

In addition, our work provides strong theoretical guarantees on the runtime of our delay controllability checker, and accordingly, we supplement them with some empirical results demonstrating that the delay controllability checking algorithm performs well in practice (Figure 9).

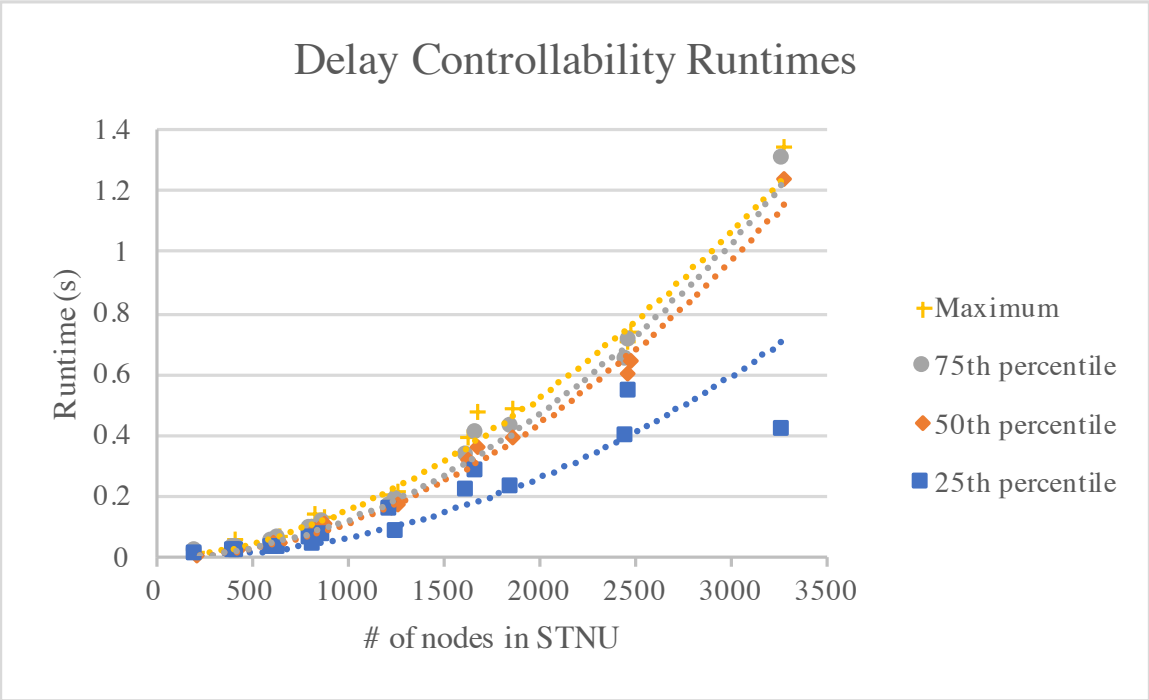

Figure 9: Runtime of delay controllability checkers on STNUs of different sizes. Shown are the 25th, 50th, 75th percentile, and maximum runtimes for STNUs of each size.

To evaluate our algorithm's runtime, we modify the autonomous underwater vehicle (AUV) example that we used in the past to evaluate dynamic controllability algorithms (Bhargava et al., 2017). In the AUV scenario, multiple vehicles are each responsible for navigating to some sites, paramaterized by an uncontrollable duration, and spending some time collecting data, an activity whose duration is specifiable by the AUV. To extend the example for delay controllability, we randomly assigned each contingent link an observation delay of either zero or infinity, representing periodic failures in communication that may be due to issues like an AUV navigating to a communication dead zone, where it has to execute a precompiled script rather than waiting for an adjustment from a central operator. We note that larger STNUs generally resemble this canonical AUV structure, as they tend to be highly patterned in their design.

For the purposes of our experiments, we uniformly randomly sampled the time taken to nagivate between sites from the integers between 0 and 10,000, and for the time spent conducting science. For the time required to collect data, we set the lower bound to 0 and let the upper bound be specified by the prior formula. If we let u and l represent the upper and lower bounds of the previous navigation task and r be a random integer uniformly sampled between 0 and 10,000, then the upper bound for time spent conducting science is specified by $\max(u - l + r - \frac{10{,}000}{v \cdot a}, 0)$, where v and a are the number of vehicles and number of activities per vehicle, respectively. Without the $r - \frac{10{,}000}{v \cdot a}$ term, the given durations allow a schedule to be precomputed, as the STNU would be strongly controllabe. By adding in

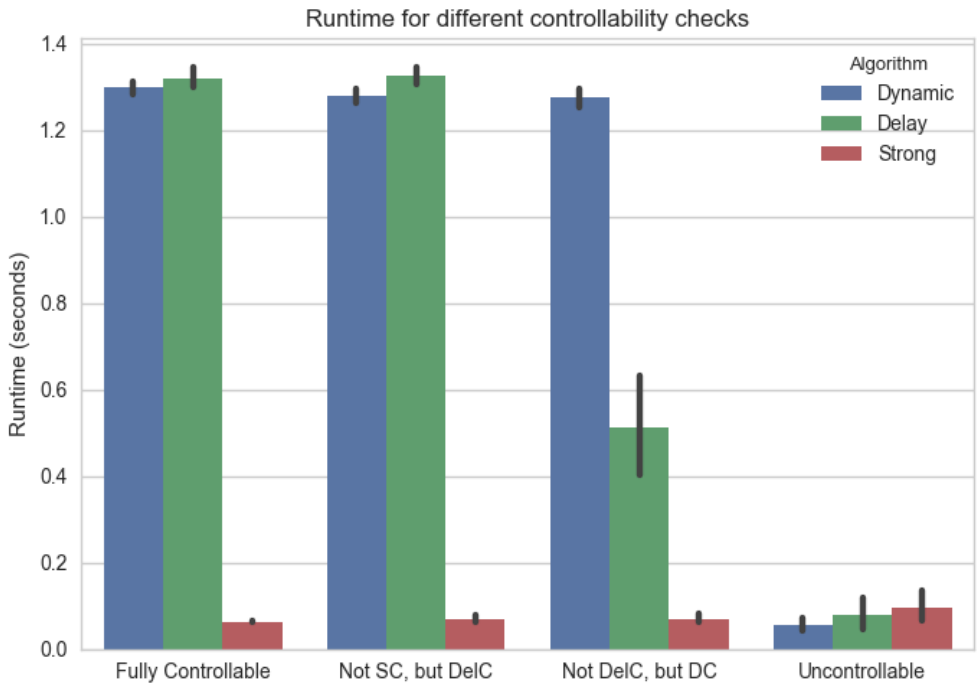

Figure 10: Runtimes of dynamic, delay, and strong controllability checkers on large STNUs. The runtimes are split based on the controllability of the network. DelC stands for delay controllability, DC stands for dynamic controllability, and SC stands for strong controllability.

the correction term, in expectation it is infeasible to complete one of the scheduled activities if the corresponding navigation task is unobserved, allowing us to get a reasonable mix of controllable and uncontrollable networks. There is an additional global temporal constraint enforcing that each AUV must finish its activities within $5000a$ units of time to simulate mission-wide deadlines.

We ran 50 trials with each of 10, 20, 30, and 40 different vehicles as well as with 10, 20, 30, and 40 activities per vehicle for a total of 800 trials.

Our results indicate that our delay controllability checker is quite efficient in practice. Most actual temporal network models have well fewer than 1000 nodes, meaning that for most purposes delay controllability checking can be used as an efficient subroutine in larger planning processses. Further, as the network size grows even larger, our runtimes still remain within a reasonable range for one-off controllability checks.

However, in order to truly evaluate the empirical performance of our delay controllability algorithm, it is important to compare the speed of delay controllability checking to the speed of checking other forms of controllability.

To investigate, we examined the relative performance of our AUV examples with 40 vehicles and 40 activities per vehicle, observing the runtime of dynamic, delay, and strong controllability algorithms (Figure 10). In order to get a sense of how performance is likely to differ across different examples, we split our results based on whether the underlying STNU was strongly controllable, not strongly controllable but still delay controllable, not delay controllable but still dynamically controllable, or entirely uncontrollable.

Across all networks, strong controllability checks complete quickly but of all the types of controllability, strong controllability checking is the most conservative. When a delay controllability check and dynamic controllability check return the same answer, delay controllability checking incurs a slight overhead, but when they differ, we see that delay controllability returns much more quickly. On their own, these results are as expected, and they mirror the difference in asymptotic complexity across the algorithms. But from an empirical perspective, they offer additional insight into how we may be able to combine the algorithms to achieve better results in practice.

From a practical perspective, the differences in performance across different situations encourage us to take a portfolio approach to solving controllability problems. Strong controllability checks are subject to false negatives, but especially on large networks, strong controllability checks finish much faster than delay controllability checks. In the event that an STNU is strongly controllable, checking strong controllability first saves a massive amount of time, whereas in the event that the network is not strongly controllable but is delay controllable, we only add a small overhead, implying that in certain scenarios, it may be advantageous to check for strong controllability before performing a full delay controllability check.

We can perform a similar analysis for dynamic controllability. In instances where a network is both delay and dynamically controllable, the dynamic controllability check presents a minor runtime improvement. However, dynamic controllability checks are subject to false positives when used as a substitute for delay controllability, and in those cases, we see a drop-off in performance. Directly checking delay controllability in instances where the problem is dynamically controllable but not delay controllable yields a runtime that is more than twice as fast. From the perspective of implementing a mixed-algorithm strategy, the large overhead in the false positive case coupled with the minor differences in all other cases imply that there is little to no advantage in checking dynamic controllability before checking delay controllability.

8. Conclusion

In this paper, we formally introduced delay controllability and showed how, with appropriate choices of our delay function γ, we can define dynamic and strong controllability in terms of delay controllability. We then provided an $O(n^3)$ algorithm that is capable of determining delay controllability, demonstrating a fundamental equivalence between dynamic and strong controllability, and then continued by evaluating the empirical attributes of delay controllability, showing that it provides a level of expressivity beyond dynamic and strong controllability while still being fast enough to use in practice.

Our efforts in this paper have been focused on establishing a more generalized notion of controllability, but there are still many promising avenues of research along these lines to

consider. One practical application of delay controllability is in characterizing the delay in propagation of information in multi-agent settings, but transmission delay is just one of the many aspects that make multi-agent coordination difficult. Understanding how communications are batched, how to handle information loss, and how to deal with uncertain and unreliable communication are all exciting future directions that build heavily on our work.

Another area of interest the generation of low-cost delay functions that still provide a guarantee of controllability over a given network of temporal constraints. Agents often have flexibility around when they communicate certain events but may prefer to delay their communication in order to minimize overhead and add convenience, as specified by some objective function. While our work speaks to how you can evaluate a particular communication protocol, it does not provide an immediate way to efficiently choose the best low-cost delay function that still guarantees controllability.

Acknowledgments

This research was funded in part by the Toyota Research Institute under grant number LP-C000765-SR.

References

Bhargava, N., Vaquero, T., & Williams, B. (2017). Faster conflict generation for dynamic controllability. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, (IJCAI-2017)*, pp. 4280–4286.

Bit-Monnot, A., Ghallab, M., & Ingrand, F. (2016). Which contingent events to observe for the dynamic controllability of a plan. In *International Joint Conference on Artificial Intelligence (IJCAI-16)*, pp. 3038–3044.

Combi, C., Hunsberger, L., & Posenato, R. (2013). An algorithm for checking the dynamic controllability of a conditional simple temporal network with uncertainty. In *Proceedings of the 5th International Conference on Agents and Artificial Intelligence (ICAART-2013)*.

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

Dechter, R., Meiri, I., & Pearl, J. (1991). Temporal constraint networks. In *Artificial intelligence*, Vol. 49, pp. 61–95.

Hunsberger, L. (2016). Efficient execution of dynamically controllable simple temporal networks with uncertainty. *Acta Informatica*, *53*(2), 89–147.

Hunsberger, L., & Posenato, R. (2016). Checking the dynamic consistency of conditional simple temporal networks with bounded reaction times. In *Proceedings of the Twenty-Sixth International Conference on International Conference on Automated Planning and Scheduling (ICAPS-2016)*, pp. 175–183.

Moffitt, M. D. (2007). On the partial observability of temporal uncertainty. In *Proceedings of the National Conference on Artificial Intelligence*, Vol. 22, pp. 1031–1037.

Morris, P. (2006). A structural characterization of temporal dynamic controllability. In *International Conference on Principles and Practice of Constraint Programming (CP-2006)*, pp. 375–389. Springer.

Morris, P. (2014). Dynamic controllability and dispatchability relationships. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pp. 464–479. Springer.

Morris, P. H., & Muscettola, N. (2005). Temporal dynamic controllability revisited. In *The Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pp. 1193–1198.

Nilsson, M., Kvarnström, J., & Doherty, P. (2013). Incremental dynamic controllability revisited. In *Proceedings of the International Conference on International Conference on Automated Planning and Scheduling (ICAPS-2013)*, pp. 337–341.

Nilsson, M., Kvarnström, J., & Doherty, P. (2014). Incremental dynamic controllability in cubic worst-case time. In *21st International Symposium on Temporal Representation and Reasoning (TIME-2014)*, pp. 17–26. IEEE.

Shah, J. A., Stedl, J., Williams, B. C., & Robertson, P. (2007). A fast incremental algorithm for maintaining dispatchability of partially controllable plans. In *Proceedings of the International Conference on International Conference on Automated Planning and Scheduling (ICAPS-2007)*, pp. 296–303.

Stedl, J., & Williams, B. C. (2005). A fast incremental dynamic controllability algorithm. In *Proceedings of the ICAPS Workshop on Plan Execution: A Reality Check*, pp. 69–75.

Vidal, T., & Fargier, H. (1999). Handling contingency in temporal constraint networks: from consistency to controllabilities. *Journal of Experimental & Theoretical Artificial Intelligence*, *11*(1), 23–45.