Multi-Agent Coordination under Limited Communication

by

Nikhil Bhargava

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2020

© Massachusetts Institute of Technology 2020. All rights reserved.

Signature redacted

Author . . . .
Department of Electrical Engineering and Computer Science
January 30, 2020

Signature redacted

Certified by . . . .
Brian C. Williams
Professor of Aeronautics and Astronautics
Thesis Supervisor

Signature redacted

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
MAR 13 2020
LIBRARIES
ARCHIVES

MITLibraries

77 Massachusetts Avenue
Cambridge. MA 02139
http://libraries.mit.edu/ask

DISCLAIMER NOTICE

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort possible to provide you with the best copy available.

Thank you.

The images contained in this document are of the best quality available.

Multi-Agent Coordination under Limited Communication

by

Nikhil Bhargava

Submitted to the Department of Electrical Engineering and Computer Science
on January 30, 2020, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

In this thesis, we present a theory for constructing real-time executives that can reason about communication between agents. In multi-agent coordination problems, different agents have different beliefs about the state of the world that can eventually be reconciled if the agents are able to share sufficient information with one another. When communication is limited, this task becomes more difficult. To achieve robustness, coordination decisions need to be made, executed, and adapted in real-time by a real-time executive.

Most existing real-time executives rely on perfect knowledge of the state of the world, making it difficult to use them in scenarios where agents either cannot or prefer not to communicate and share information. This thesis offers three contributions that together provide the basis for constructing a real-time executive capable of handling multi-agent coordination under limited communication.

First, we introduce delay controllability as a way to augment the input plan representation to including a communication model. Delay controllability lets us reason about multi-agent activities under limited communication in the form of communication delays and provides a guarantee that problem evaluation is tractable.

Second, we provide a way to indicate by when each agent must communicate the results of their actions. Many agents have flexibility in choosing exactly when to communicate. We provide an algorithm for choosing a low-cost set of moments to communicate and present a strategy for adjusting those strategies when communication networks are unreliable causing disruptions in the original communication plan.

Third, we offer a way to model noisy communication. Noisy communication offers approximate temporal information that is useful during execution but is generally difficult to incorporate. We introduce variable-delay controllability as a way to model this kind of communication and provide the first sound and complete algorithm for incorporating noisy information that runs in polynomial time.

Thesis Supervisor: Brian C. Williams
Title: Professor of Aeronautics and Astronautics

Acknowledgments

Nobody does a Ph.D. alone, and I've been extremely lucky to have such an incredible group of people who have supported me, given me advice, and helped me along the way.

First, I want to thank my advisor, Professor Brian C. Williams. Prof. Williams took a gamble on a young student without a real academic track record or a strong idea of where to go. In my first year here, Prof. Williams was instrumental in shaping my thesis direction and helping me discover the area I wanted to work in. Since then, Prof. Williams has been an amazing research partner, helping me focus my efforts on the right problems and magnifying my impact. I would also like to thank my committee members, Professor Randall Davis and Professor Tomás Lozano-Pérez. Their inputs and guidance throughout the thesis process were critical in helping me develop my work to where it is today.

I would also like to thank all the members of the MERS group, both past and present. MERS has been my home at MIT since I arrived, and I have been blessed to have such an amazing group of peers. The members of MERS are smart, kind, and thoughtful people; such a combination in such a large group is rare to find. It was an honor to be part of such an amazing team. Thank you Allen, Andrew, Ashkan, Ben, Charles, Christian, Cyrus, Dan, Enrique, Eric, Jacob, Jingkai, Matt, Matthew, Meng, Nick, Peng, Sang, Sylvia, Simon, Steve, Sungkweon, Tiago, Yang, Yuening, Yui, and Zach for being an important part of my MIT experience.

I also want to thank my friends and family for being a constant source of support for me, and for undertaking the difficult job of helping me stay sane. In particular, a special thanks goes to Ben, Oana, Michael, Sophie, Arjun, Irene, Matt, Amanda, Ellora, Yoseph, and all of the other folks in Boston for their willingness to indulge me in the many, many moments where I grew tired of working. In particular, I want to thank my girlfriend Dana for being an amazing partner and helping me handle some of my proudest and most frustrating moments in grad school. I'm excited for what the future has in store.

It also goes without saying that I owe so much to my parents, Gautam and Reena, and my sister, Zoe. They have supported and loved me unconditionally, and I would not be here today if not for them. They pushed me to chart out my own course and to seek happiness, and to this day, I continue to turn to them for advice and wisdom. Thank you for everything.

Finally, this work was partially supported by the Toyota Research Institute (TRI). However, this thesis solely reflects the opinions and conclusions of its author and not TRI or any other Toyota entity.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Understanding multi-agent systems and how they operate is at the heart of many problems in artificial intelligence. These problems at their core involve an automated system reasoning and making decisions in a way that deeply affects the actions and goals of other agents, be they human or robotic. This thesis aims to examine multi-agent planning and execution in the temporal domain and to provide a robust theory that enables improved plan execution when communication between agents is limited.

This chapter provides a high-level overview of the contributions that this thesis will provide. It starts by presenting a series of motivating examples, illustrating both the ubiquity of and the challenges present in multi-agent coordination problems. This chapter then describes the overall approach taken to augmenting a system capable of reasoning about and providing plans for multi-agent coordination problems. The rest of this thesis provides a desiderata, describing a set of requirements for an effective multi-agent coordination executive, a model that satisfies those requirements, and a series of algorithms which allow that model to be used efficiently in practice.

1.1 Motivating Examples

We take as an initial motivating example, the problem of autonomous vehicle dispatch. As autonomous vehicles become more advances, car companies are beginning to envision a future where they move from being manufacturing companies to mobility and transportation companies. In a future where autonomous cars are ubiquitous, companies will survive and excel based on the quality of their service.

When transitioning to transportation companies, car companies have to efficiently and effectively solve a particular type of multi-agent scheduling problem, where a single central agent, the autonomous car company, is responsible for dispatching cars to serve the needs of all of its customers. Each customer has an internal set of deadlines and goals that is only made available to the dispatcher in limited quantities (e.g. a customer may request a car to 42 Vassar St. at 3:45pm, but the dispatcher does not know if its purpose is a business meeting or coffee with a friend). Despite the fact that communication between customers and the autonomous dispatcher is limited, these companies must still construct policies and schedules that guarantee high-quality service.

The problem that these car companies face is the focus of this thesis. In order to develop a high quality system, they must be able to reason about the temporal goals and requirements of their customers under limited overall communication. This example also hints at another important requirement of algorithms that reason over multi-agent coordination problems. In order for them to be used in situations like autonomous car dispatch, it is of critical importance that these algorithms operate efficiently. It is plausible, and at times likely, that such a system would be responsible for accommodating tens of thousands of requests in a single interval. When discussing this thesis's contributions, we will make sure to speak to the efficiency of the presented algorithms in practice.

While the autonomous car scenario represents one future problem that this thesis may help address, there are similar analogs in the here and now for which multi-agent coordination under limited communication is well-suited.

Consider the following example about the deployment of autonomous underwater vehicles (AUVs).

Example 1.1. Autonomous Underwater Vehicle Operation

The Woods Hole Oceanographic Institution (WHOI) is interested in using

> a heterogeneous set of three autonomous underwater gliders to explore areas of the seabed. The gliders are designed to be incredibly energy-efficient in their traversal, meaning they can operate autonomously for days and even weeks without human intervention. Some degree of coordination is necessary across the vehicles. Each of the vehicles has different sensors on-board, so each may need to independently traverse the same region. However, to avoid collision, the vehicles need to coordinate to ensure they are not operating in the same regions simultaneously.
>
> Unfortunately, direct communication between gliders is impossible, and communication between a glider and shore operations only happens when that glider has surfaced. While it might seem reasonable to surface the glider before and after every action, there is a direct trade-off between time spent surfaced and amount of time spent conducting scientific operations.

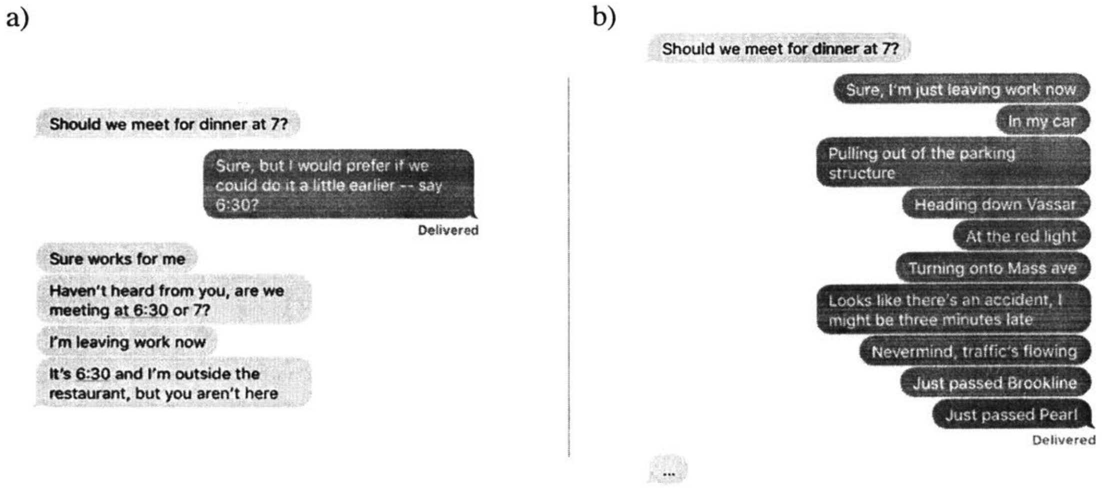
Beyond the operational difficulties in choosing high-value sites and dealing with temporal uncertainty in the form of uncertain currents, what this example illustrates is that considerable thought needs to go into selecting times to communicate and to verify that those choices lead to a safe and efficient plan.

This example is not meant to be purely theoretical. Prior collaborations between WHOI and the Model-based and Embedded Robotics System (MERS) group at MIT focused on deploying vehicles to solve these kinds of problems [1, 53], and in November 2019, MERS and WHOI collaborated again to demonstrate exactly this capability using the concepts outlined in this thesis.

Even in simpler instances, such as those involving human agents planning an outing, we see the need for a theory that characterizes how to act under limited communication. Consider the following simple example.

Example 1.2. Alex and Sam get dinner.

> Two agents, Alex and Sam, want to meet at a restaurant and eat dinner together.



a)

Should we meet for dinner at 7?

Sure, but I would prefer if we could do it a little earlier -- say 6:30?

Sure works for me

Haven't heard from you, are we meeting at 6:30 or 7?

I'm leaving work now

It's 6:30 and I'm outside the restaurant, but you aren't here

b)

Should we meet for dinner at 7?

Sure, I'm just leaving work now

In my car

Pulling out of the parking structure

Heading down Vassar

At the red light

Turning onto Mass ave

Looks like there's an accident, I might be three minutes late

Nevermind, traffic's flowing

Just passed Brookline

Just passed Pearl

Figure 1-1: a) An example multi-agent plan execution run that failed due to under-communication. b) Over-communication in the context of multi-agent plan execution.

It is immediately clear to the observer that constructing a plan for this problem is straightforward. The domain has a highly constrained action space, the degree of interactions between agents is low, and all agents are working collaboratively towards the same shared goal. Applying planning algorithms to this scenario feels like overkill because as humans, solving these kinds of problems is routine.

Nonetheless, it is possible for multi-agent coordination to fail in execution. Many of us have been in exactly this scenario where plans fail to materialize due to a breakdown in communication (see for example Figure 1-1a). It is clear that for many types of multi-agent plans to be successful, there has to be a baseline level of coordination and communication during plan execution for a plan to be successful.

Of course, a simple solution is to over-communicate and to share all possible pieces of information that may be relevant to the joint plan (see for example Figure 1-1b). While this approach provides a theoretical guarantee of success, the dispatch of information itself can come at a cost or be highly distracting for the message recipient. An appropriate multi-agent plan execution system, therefore, must be capable of being judicious in its communication to strike a balance between not providing enough information and overwhelming the end-user.

As human agents, we intuitively recognize when communication strategies are suboptimal but sometimes struggle to find good ones. Managing communication is a critical part of ensuring the success of multi-agent plans, and this thesis will provide a set of tools to describe how to do so efficiently and effectively.

1.2 Approach

This thesis will address the multi-agent plan execution problem by providing the theory and corresponding algorithms for a multi-agent real-time executive.

At its core, a real-time executive operates in three stages (see Figure 1-2a for details). As its initial input, a real-time executive is provided with an ungrounded temporal plan and action model where the timings of specific actions are yet to be made. The job of the executive is to ground that schedule and dispatch those actions to agents in the world who are capable of carrying them out. In a world where all agents behave deterministically and there is no uncertainty, these capabilities alone are sufficient. However, in practice, agents do not always execute tasks exactly when specified, and some events are outside the purview of the executive. In those situations, the executive needs to receive updates from the agents about the effects of the dispatched actions and the updated state of the world. The executive then incorporates that information into its internal representation and amends its subsequent dispatches.

While many executives have been developed in the past [20, 34, 35], they do not tend to be well-suited to multi-agent plan execution when communication between agents is limited. In this thesis, we will improve the presented real-time executive architecture by showing how to modify each component of the real-time executive architecture to better handle the nuances of multi-agent plan execution (see Figure 1-2b).

First is augmenting the input. In this thesis, we will primarily concern ourselves with ungrounded temporal plans, meaning the responsibility of a real-time executive is to schedule the dispatch of and subsequent communication about the results of a

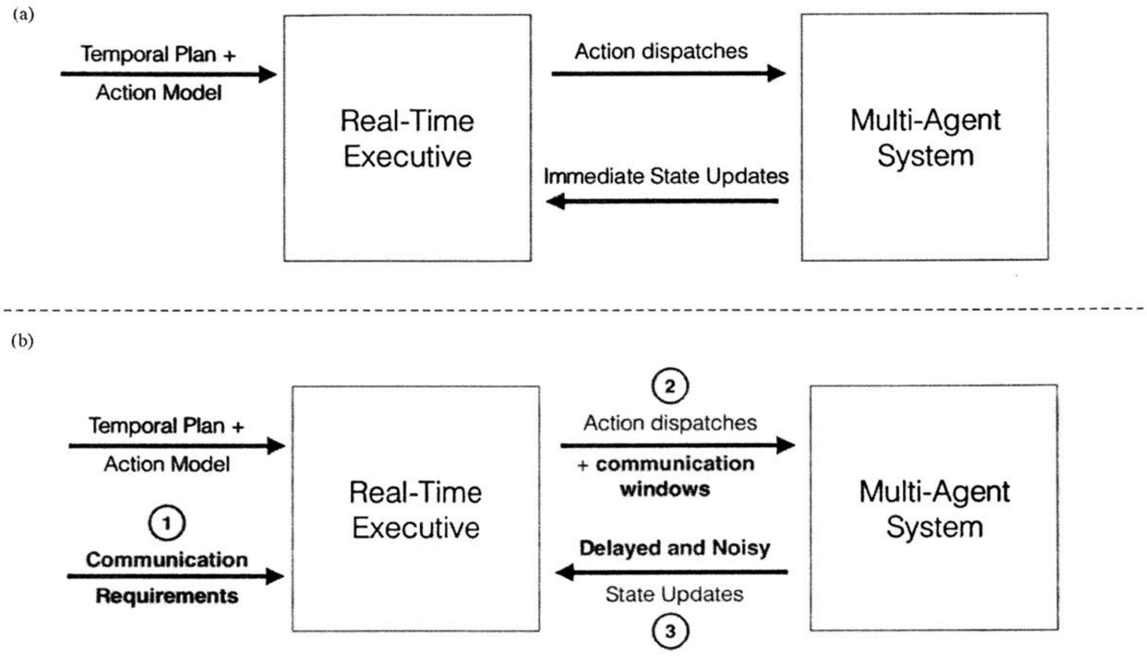


Figure 1-2: a) A typical architecture diagram for a temporal-based executive. b) An architecture for a temporal executive enabled by my research which incorporates asynchronous and uncertain communication.

set of pre-determined actions.

In order to arrive at our executive's input representation, we can start from any of a large number of domain independent languages that are used for planning. In particular, the Reactive Model-Based Programming Language (RMPL) [34], Planning Domain Definition Language (PDDL) [26], and Temporal Concurrent Automata (TCA) [31] specifications all provide higher-order semantics for defining actions broadly in terms of their preconditions and effects. When time is involved, the resulting representations can often be reduced down to Simple Temporal Networks (STNs) [22], Simple Temporal Networks with Uncertainty (STNUs) [58], or some related extension; we will primarily consider STNUs in the context of this thesis and provide a rigorous discussion of the possible alternatives in Appendix A.

Our interest is in constructing an executive that reasons over and is able to execute a plan efficiently while addressing the difficulties of multi-agent communication. To adequately handle this, we introduce a new formalism, delay controllability, which models delays in communication as layered on top of an STNU and show that we can reason about and execute a plan subject to delays in communication in polynomial time. Delay controllability forms the backbone of the rest of the work in this thesis.

Second, we update the executive to include communication windows for actions alongside action dispatches. Delay controllability algorithms evaluate whether it is possible to execute a given temporal plan subject to communication delays associated with a given communication strategy. However, many agents have flexibility in deciding when to communicate and have soft preferences about when communication happens based on cost and convenience. The goal of including communication windows is to establish what information must be shared for plan success while giving agents flexibility as to exactly when they report on their progress. We call the problem of determining the best possible communication windows the Communication Cost Minimization Problem (CCMP).

What makes solving CCMPs difficult is that the space of communication strategies is continuous and unbounded. To address this problem we borrow ideas from conflict-directed search [60] and understand that if we know why a particular communication strategy is infeasible, we can adjust future strategies accordingly. As with most search based algorithms, there is a trade-off between empirical performance and optimality guarantees. We show that Least Cost Resolution Search in practice is significantly faster than an optimal search algorithm and provides near optimal results, even if it is possible to construct adversarial examples for which the algorithm gives polynomially bad approximations.

Our work in this area also considers how to adjust communication windows when information comes in earlier or later than expected. When dispatching across real agents, there is always some risk of action failure. Understanding how to adjust plans in those instances is critical to improving the chance of plan success.

Third and finally, we focus on making the executive tolerant to noisy and imprecise updates from agents. When dealing with agents in the real world, their communication and utterances may not always be entirely accurate. For example, when a human agent says "I finished lunch at 1pm," they likely mean they finished at 1pm ±

10 minutes. Even in scenarios exclusively filled with robotic agents, having tolerance for noise in communication is important to ensure overall robustness.

Here we introduce variable-delay controllability as an extension of delay controllability to account for noisy delays. In this model, agents may know that an event happened over some time period in the past, but they are not guaranteed to know exactly when it happened. This communication now acts as a mechanism to partially resolve temporal uncertainty, and the question is how to use that information for improved execution. We show how to reduce variable-delay controllability to general delay controllability in the case where observational uncertainty is set-bounded. When we have a large probabilistic distribution over possible observations and we tolerate some small risk of failure, we introduce the notion of chance-constrained variable-delay controllability. In such situations, we show how to interleave the use of a non-linear program solver with delay controllability conflict extraction techniques to determine whether a temporal plan can still be executed.

1.3 Contributions

This thesis provides a framework for reasoning about and evaluating multi-agent coordination under limited communication. The contributions can be subdivided into three parts.

1. **A formalism for reasoning about multi-agent execution.** Different types of temporal formalisms can encode multi-agent planning and execution problems with varying levels of fidelity. In Appendix A, we examine many of these formalisms in detail, providing novel hardness and completeness results for many types of these bounds.[1] In Chapter 3, we introduce delay controllability as a means of reasoning about multi-agent scenarios efficiently through the lens of communication delay and in Chapter 4 show how to check delay controllability in polynomial time.[2]

[1] Much of this work was originally presented in AIJ [8] and AAMAS [7].

[2] This work was originally presented as a technical report in DSpace@MIT [3].

2. **An algorithm to derive required communication windows.** Delay controllability as presented only answers the binary decision problem of whether it is possible to reactively construct a schedule given a communication strategy. Agents, however, often have flexibility in choosing when to communicate, with the understanding that some times may be more inconvenient or costly for communication than others. We encode this problem as a Communication Cost Minimization Problem (CCMP). In Chapter 5, we present a series of algorithms for constructing low-cost communication window solutions to CCMPs as well as a description for how to adapt those strategies online during execution. We show that while certain sub-optimal algorithms may yield polynomially bad approximations in adversarial settings, in practice they are significantly faster and provide results that are near optimal.[3]

3. **A procedure for handling noisy communication.** The assumptions made to this point rely on the belief that while communication may be delayed, it is always accurate. In Chapter 6, we introduce variable-delay controllability as a means of handling temporal noise in communication events. We show how to reduce a variable-delay controllability problem to that of ordinary delay controllability in linear time and introduce a procedure for reasoning over probabilistic noise through chance-constrained variable-delay controllability, providing experimental results demonstrating that these approaches can be used efficiently in practice.[4]

1.4 Organization of Thesis

The rest of the thesis is organized as follows. In Chapter 2, we provide some preliminary background and notation that we will use across the rest of the thesis. In Chapter 3, we provide a desiderata, which describes the ideal contributions of a temporal modeling framework for multi-agent coordination, and a set of definitions which

[3] The first part of this work was originally published in IJCAI [4].

[4] The first part of this work was also originally published in IJCAI [5].

satisfy its corresponding requirements and form the cornerstone of this thesis's contributions. Chapter 4 provides an efficient algorithm for modeling communication delays for multi-agent plan execution through the lens of delay controllability. Chapter 5 builds on the work of the previous chapter, extending core delay controllability algorithms to make them suitable for enumerating solutions to CCMPs. In Chapter 6, we consider how to reason about uncertain and imprecise communication during execution with variable-delay controllability and provide a procedure for doing so efficiently when given both set-bounded and probabilistic distributions over the uncertainty in communication. Finally, in Chapter 7, we provide some concluding remarks and elaborate on promising directions for future research endeavors.

Chapter 2

Background

In this chapter, we provide the required preliminary background material that will be used across the rest of the thesis. Some discussion and exposition in this chapter has been included from previous writings from the thesis author [3].

2.1 Simple Temporal Networks

Simple Temporal Networks (STNs) are the most basic temporal network on which other temporal network formalisms are built [22]. STNs are composed of a set of variables and a set of binary constraints limiting the difference between any two variables (e.g. $B - A \in [10, 20]$). These variables denote individual points in time (henceforth *events*) and the constraints between them are binary temporal constraints, limiting their temporal difference (e.g. event A must happen between 10 and 20 minutes before event B).

Definition 2.1. STN [22]
An STN is a pair $\langle X, R \rangle$ where:

- X is a set of event variables, whose domains are the reals
- R is a set of constraints. Each constraint has scope $x_r, y_r \in X$ and relation of the form $x_r - y_r \in [l_r, u_r]$

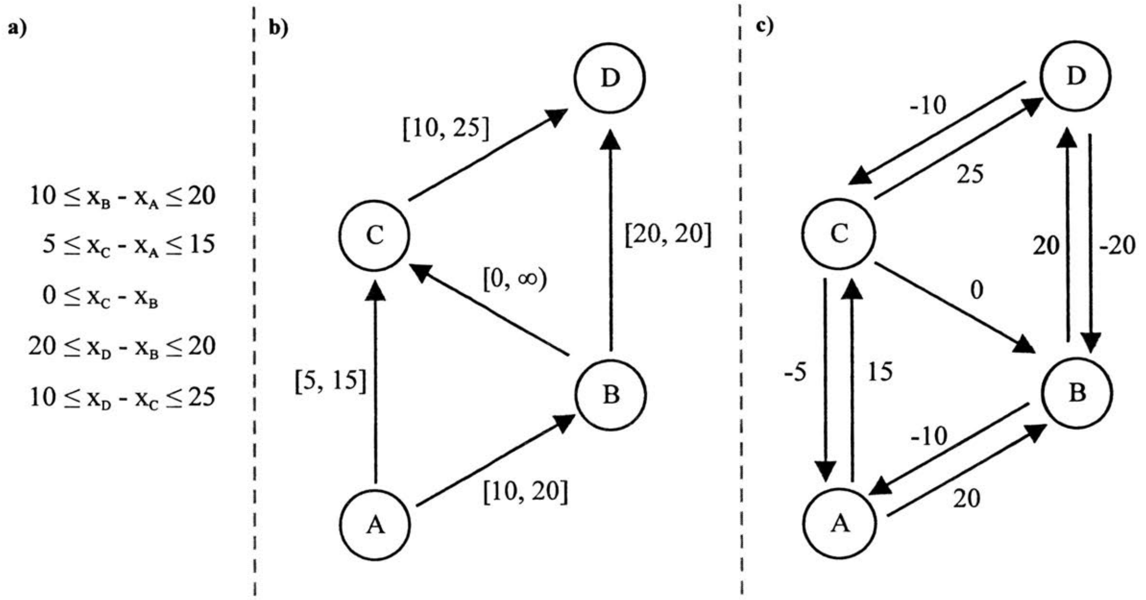


Figure 2-1: (a) An STN as specified by its set of constraints. (b) The same STN represented graphically. (c) The same STN represented using its distance graph formulation.

An STN denotes a set of linear inequalities (Figure 2-1a) that are visualized as a graph, where each event is represented as a node and edges between nodes represent constraints (see Figure 2-1b).

In the context of this thesis, the STN represents the base model that we use to construct a theory of multi-agent coordination under limited communication. Because our focus is on the execution of tasks subject to temporal deadlines, STNs are used to indicate the possible durations and dependencies between actions in our desired contexts.

When we consider the feasibility of an STN, we are generally concerned with whether it is possible to construct a *schedule*, that is an assignment of values from $\mathbb{R}$ to each event $x \in X$, such that all constraints are satisfied. An STN is *consistent* if there exists a schedule for all events such that all constraints are respected.

To evaluate consistency, we often reason over an STN's *distance graph* (see Figure 2-1c). A distance graph is like an STN, in that each event becomes a node, while each constraint, $u_i \leq x_j - x - K \leq v_i$, becomes two edges – one in the original direction with weight v_i and one in the reverse direction with weight $-u_i$ (see Figure 2-1c).

From previous work, we know that an STN is consistent if and only if there is no negative cycle in its equivalent distance graph [22]. In general, we take n to be the number of events in a temporal network and m to be the number of constraints. Checking the consistency of an STN is doable in $O(mn)$ time, using Bellman-Ford to check for negative cycles. In Appendix B, we provide an alternative explanation for why finding negative cycles is sufficient to prove STN inconsistency.

2.2 Simple Temporal Networks with Uncertainty

While the STN formalism is quite powerful, it has one major shortcoming; it assumes that the scheduler has absolute control over each and every event. However, there are many scenarios where the scheduler does not have this kind of absolute control and where this uncertainty needs to be addressed. For example, individuals are unable to control how much traffic will affect their morning commute or when it might start to rain. Moreover, they are quite unlikely to have precise control of other agents in a multi-agent environment. By convention, we say that those events and actions not explicitly chosen by the scheduler are chosen by nature.

To account for this, we need a way to augment temporal networks to describe the uncertainty often found in temporal processes. Simple Temporal Networks with Uncertainty (STNUs) extend STNs, allowing us to model events whose timings are outside the control of the scheduler [58].

Definition 2.2. STNU [58]

An STNU is a 4-tuple $\langle X_e, X_c, R_r, R_c \rangle$ where:

- X_e is the set of executable events
- X_c is the set of contingent events
- R_r is the set of requirement constraints of the form $l_r \leq x_r - y_r \leq u_r$, where $x_r, y_r \in X_c \cup X_e$ and $l_r, u_r \in \mathbb{R}$
- R_c is the set of contingent constraints of the form $0 \leq l_r \leq c_r - e_r \leq u_r$, where $c_r \in X_c$, $e_r \in X_e$ and $l_r, u_r \in \mathbb{R}$

In STNUs, events are subdivided into executable and contingent events and constraints are subdivided into requirement and contingent constraints. Executable events are the events that the scheduler is responsible for (and are equivalent to events in STNs), whereas contingent events are scheduled by nature. Requirement constraints are equivalent to ordinary STN constraints and are free to constrain any pair of events. Contingent constraints, in contrast, represent relations between a starting executable event and an ending contingent event that nature is guaranteed to enforce.

By convention, the lower-bound of a contingent constraint is required to be non-negative to enforce that the ending event of the constraint follows the starting event. It is worth noting that we require that all contingent constraints begin from an executable event. This requirement does not have an impact on the expressiveness of our networks but will simplify further discussions in this thesis. It is simple to take a pair of chained contingent constraints and splice in a new executable event that starts the second contingent constraint and is required to occur at the same time as the first contingent constraint's end event.

To expand on this, in order to simplify our reasoning around STNUs, we will make the assumption that all STNUs are in *normal form.* To transform an STNU into normal form, we replace all contingent constraints $A \xRightarrow{[x,y]} C$ by introducing a new event A' with two constraints, $A \xrightarrow{[x,x]} A'$ and $A' \xRightarrow{[0,y-x]} C$. From an expressiveness perspective, the two STNUs are equally expressive, meaning the transformation does not affect delay controllability.

For a contingent constraint r, represented as $l_r \leq c_r - e_r \leq u_r$, we say that the *duration* of r is the value specified by the difference $c_r - e_r$ at execution time. When considering the interplay between the scheduler and nature, we are free to describe nature's role as either picking the durations of contingent constraints or as picking the times of each contingent event. The set of contingent constraint durations together with the set of executable events uniquely determines a set of contingent events, indicating that the two are equivalent.

Since contingent events have unknown assignments in an STNU, it is difficult to reason directly about the consistency of an STNU, as we need some way to quantify over or otherwise characterize the uncertainty of contingent events. For STNUs, we care about evaluating their *controllability,* or whether it is possible to provide an assignment to all executable events in response to some observation of contingent events. Historically, STNU controllability has come in three forms: strong, weak, and dynamic [58].

2.2.1 Strong Controllability

We say that an STNU is strongly controllable if there exists some schedule for all executable events X_e, such that for every possible assignment of values to contingent events in X_c that satisfy the contingent constraints R_c, all of the requirement constraints R_r are satisfied. STNU strong controllability checking, much like STN consistency checking, reduces to detecting the presence of a negative cycle and can be computed in $O(mn)$ time [58].

While strong controllability provides extremely strong guarantees with its fixed schedule, requiring strong controllability tends to be quite conservative in practice, as it precludes the ability of the scheduler to react to any observations of contingent events.

2.2.2 Weak Controllability

Weak controllability asks whether it is possible to reactively construct a schedule if the durations of the uncertain events are revealed before scheduling begins. In other words, for every fully specified set of contingent action durations that guarantee satisfaction of contingent constraints R_c, weak controllability asks whether it is always possible to pick a set of values for the executable events X_e such that all requirement constraints R_r are satisfied. While checking whether a schedule exists for any one particular realization of the uncertain events reduces to checking STN consistency, checking STNU weak controllability in general is coNP-complete [39].

Weak controllability is a highly reactive mode of constructing a schedule. It assumes that all information is made available to the scheduler before execution and considers the ability of the scheduler to react appropriately in all particular situations. In this way, weak controllability can be seen as the natural dual of strong controllability. Together these two modes map out the two extremes of constructing valid STNU schedules.

2.2.3 Dynamic Controllability

Informally, we say that an STNU is dynamically controllable if it is possible to assign values to executable events given knowledge about assignments to only those events that happened in the past, including past contingent events. While strong controllability constructs a schedule before the fact, dynamic controllability affords additional flexibility by incorporating information as it arrives to construct a valid schedule.

In some way, dynamic controllability can be seen as the intuitive composition of strong and weak controllability. Weak controllability assumes perfect foresight for future uncertain events, whereas strong controllability assumes that no events can ever be observed. Dynamic controllability blends the two by assuming full observation of events in the past while still holding unobserved events that come in the future. This combination makes it desirable for use in scheduling and execution, as it does not require the levels of foresight that weak controllability may, but at the same time it is able to take advantage of information at a greater rate than strong controllability checks may. In the next section, we provide a deeper look at dynamic controllability and how to evaluate dynamic controllability in STNUs.

2.3 Differentiating Types of Controllability

The three presented types of controllability each provide different ways of evaluating completeness. While weak controllability tends not to be used in practice (as it involves some level of clairvoyance from the scheduler), strong and dynamic controllability are used quite commonly to schedule agents in temporal problems.

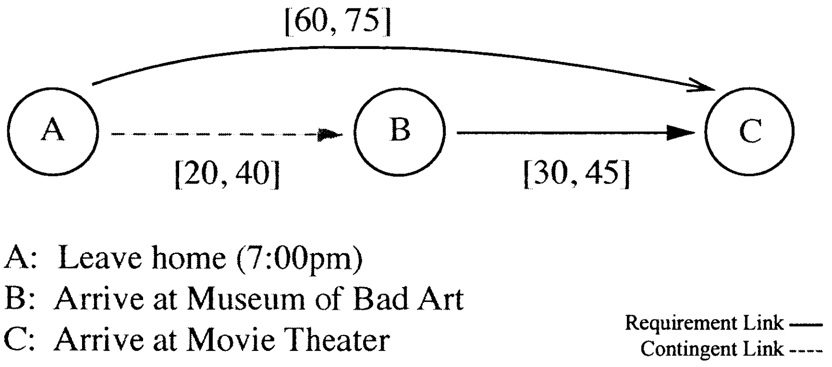


Figure 2-2: Sam goes to the Museum of Bad Art and the movies. Figure for Example 2.1.

Here, we will present a series of examples to more intuitively illustrate the differences between strong and dynamic controllability.

Example 2.1. Sam goes to the Museum of Bad Art followed by the movies. See Figure 2-2.

> Sam wants to spend 30 to 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 depending on traffic. She does not want to be more than 15 minutes early to the movie, and she definitely does not want to be late.

In this example, there are three events in total which compose the temporal network. We let event A denote when Sam leaves home, event B denote when she arrives at the Museum of Bad Art, and event C denote when she heads upstairs to the movie theater. For convenience, we assume that event A is fixed and always occurs at 7pm.

Example 2.1 is not strongly controllable in that there is no way to commit to a schedule for the three events ahead of time. If Sam decides to head upstairs from the museum at any time before 8:10pm and it takes a full 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 just 20 minutes, she will spend too much time at the Museum of Bad Art.

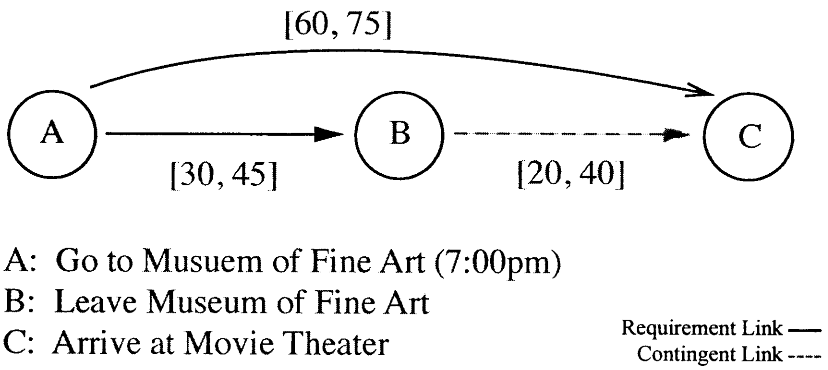

[60, 75]
A
B
C
[30, 45]
[20, 40]
A: Go to Musuem of Fine Art (7:00pm)
B: Leave Museum of Fine Art
C: Arrive at Movie Theater
Requirement Link
Contingent Link

Figure 2-3: Sam goes to the Museum of Fine Art and the movies. Figure for Example 2.2.

However, it is possible to construct a feasible 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 the problem's constraints, we can generate an example that is neither strongly nor dynamically controllable.

Example 2.2. Sam goes to the Museum of Fine Art and the movies. See Figure 2-3.

> 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 2.2 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 an 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.

Chapter 3

A Framework for Temporal Coordination with Communication Delay

In this thesis, we focus on the roles that communication and delay play in the execution of a multi-agent plan. We use this lens as a way to better model the multi-agent scheduling problem and to construct efficient algorithms for doing so. One of the major goals of this thesis is to provide a framework for modelers that allows them to describe multi-agent scenarios where communication is restricted or limited in some way. Our focus in this chapter is to lay out the set of requirements and desiderata for our framework and to introduce the specific definitions that satisfy those requirements. In Chapters 4, 5, and 6, we provide a series of algorithms for evaluating and constructing schedules for these models.

3.1 Formalizing the Requirements

Our goal in establishing desiderata is to come up with a set of features that are needed in order to augment a real-time executive to allow it to adequately handle multi-agent scenarios (see Figure 1-2). We derive an effective set of requirements by examining the component inputs and outputs of a standard executive and reasoning about what is needed to ensure that it is capable of operating correctly in the presence of communication that is delayed or absent.

First, we need to augment the input temporal and action model with a communication model to provide a way to describe delays and interruptions to communication in a more structured way. While we can choose from many possible models of communication, it is important to choose one that can distinguish between when an action occurs and when each agent learns about it. It is important that the model allows for different agents to experience different delays in their observations of actions and for delays to have some amount of variability in when they happen (we clarify later exactly how we accommodate this).

Second, our framework should be aware that certain communication is expensive and should be capable of deciding when the results of certain events should be communicated, if at all. Agents often have control over when they communicate the results of an action or even whether they communicate those results at all. In the interest of minimizing the cost of communication, whether that is in the form of preserving bandwidth, minimizing distractions, or maintaining privacy, it is important to be able to describe the cost associated with communication and formulate a problem whose aim is to minimize that cost while still guaranteeing the satisfaction of all temporal constraints. Adding this formalization allows us to determine specific communication windows that agents must adhere to.

Third, our framework must establish a way to model noise in the communication itself. Imprecise communication is a hallmark of the way that humans coordinate with one another (for example saying "I left 15 minutes ago" really means "I left somewhere between 5 and 20 minutes ago"). Humans are adept at reasoning in the face of this kind of uncertainty, and as a result, we expect our frameworks for execution to be capable of representing and evaluating this as well.

The final requirement is that the models and corresponding algorithms must be tractable. Extensions to STNUs involving disjunctive constraints [55, 57], conditional constraints [30], and even those with generalized multi-agent observation [36, 15] have been proposed as more expressive theories capable of handling richer and richer situations. Unfortunately deriving solutions for most of these models is NP-hard, and for those that are not proven NP-hard, we do not yet know whether evaluating the models is guaranteed to be tractable (see Appendix A). Ultimately, these results yield bounds that are infeasible for use in problems at scale, prompting us to search for a new alternative that satisfies the desiderata.

3.2 Introducing Delay Controllability

In this section, we describe how we build our base model, how we use it to satisfy the requirements set forth in the previous subsection, and ultimately what kinds of problems we can newly solve with these tools.

3.2.1 Delay Controllability

The cornerstone of our contributions is a generalized model of controllability, called *delay controllability.* Delay controllability concerns policies for constructing schedules that are robust to the type of uncertainty faced in situations where communication is not always reliable. Consider, for example, scheduling the activities of a pair of human agents. Each agent prefers that all scheduling constraints are satisfied and accordingly may be proactive in their communication. However, due to preoccupation, forgetfulness, or a need for privacy, we may not always have perfect knowledge of when agents executed their actions. Delay controllability aims to provide a way to incorporate information as it arrives in order to guarantee successful coordination across agents, even when that information is delayed or occasionally absent.

In reviewing other temporal network frameworks for modeling multi-agent coordination, we see that two of the main types of controllability, dynamic and strong, can be insufficient when attempting to model and ultimately schedule multi-agent communication that involve delayed communication. Strong controllability requires pre-computing a schedule, which guarantees success in the face of any delay in communication but is in practice too conservative. Dynamic controllability, in contrast, assumes that all information is freely available during execution, which is often an incorrect assumption when dealing with multiple agents.

Instead, we need a solution that lies somewhere in the middle. Understanding delays in communication is at the heart of this problem; dynamic controllability assumes that there are no delays in communication, while strong controllability assumes that delays are so overwhelming that it is not worth relying on the arrival of any information to compute a schedule. Delay controllability addresses this divide by generalizing the two forms of controllability and exposing a class of controllability problems that lie in between, by explicitly modeling what events can be observed and the delays associated with event observation.

Examples

To motivate the value of delay controllability, we introduce two new examples that differ only with respect to when information is relayed.

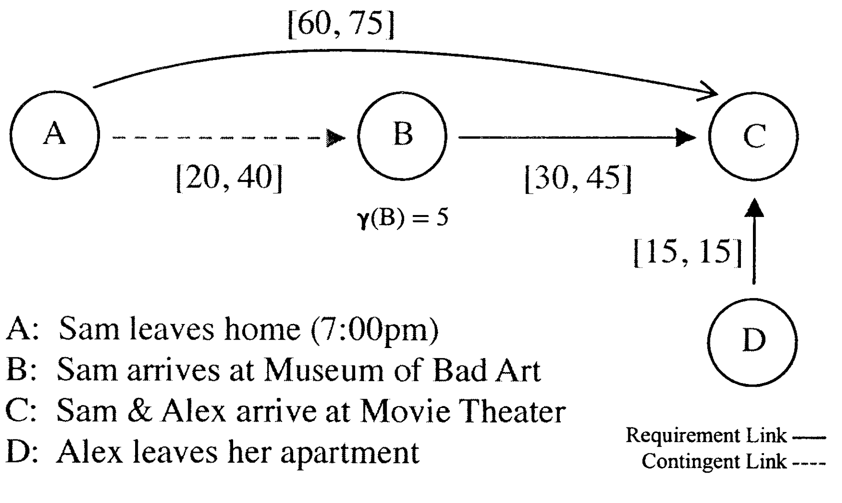

[60, 75]
A
B
C
D
[20, 40]
[30, 45]
γ(B) = 5
[15, 15]
A: Sam leaves home (7:00pm)
B: Sam arrives at Museum of Bad Art
C: Sam & Alex arrive at Movie Theater
D: Alex leaves her apartment
Requirement Link ——
Contingent Link ----

Figure 3-1: Sam and Alex go to the movies. Sam's phone needs to recharge for 5 minutes to let her call Alex. Figure for Example 3.1.

Example 3.1. Sam and Alex go to the movies. To coordinate, first Sam's phone needs to be recharged. See Figure 3-1.

> 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 decides 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 meet Alex more than 15 minutes before the movie starts, 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. They need to enter the theater at the same time because the movie is sold out, and they want to get seats together. Sam's phone needs to be recharged before calling Alex. When Sam arrives at the museum, she will leave her phone to charge at the museum front desk for 5 minutes, before calling Alex.

Example 3.1 is controllable. As was the case in Example 2.1, Sam can make decisions on the fly to ensure that she spends an appropriate amount of time at the Museum of Bad Art while guaranteeing that both she and Alex reach the theater at the same time. For example, if Sam calls Alex within 5 minutes of arriving, there will still be at least 30 minutes left before the movie starts. As a result, Alex has plenty of time to make it to the movie theater in time.

In contrast, consider Example 3.2, which only differs in that it takes 40 minutes, instead of 5 minutes, to recharge Sam's phone:

Example 3.2. Sam and Alex go to the movies, but Sam's phone has been deeply discharged. See Figure 3-2.

> 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 decides 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 meet Alex more than 15 minutes before the movie starts, and she definitely does not want to be late.

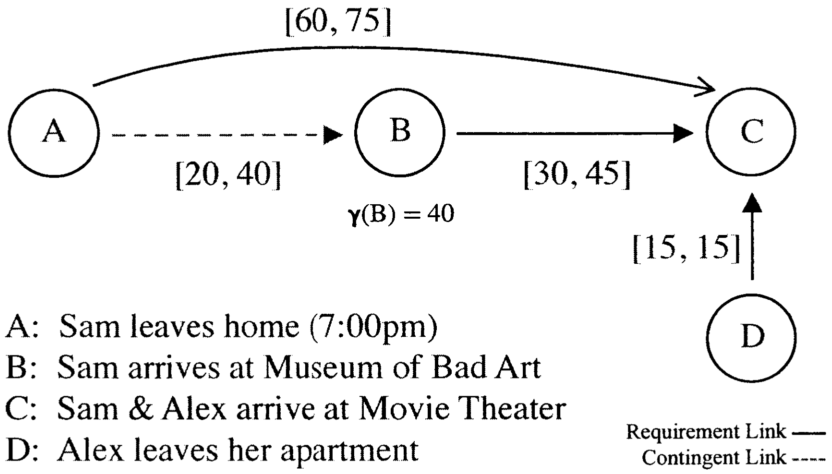

[60, 75]
A
B
C
[20, 40]
[30, 45]
γ(B) = 40
[15, 15]
D
A: Sam leaves home (7:00pm)
B: Sam arrives at Museum of Bad Art
C: Sam & Alex arrive at Movie Theater
D: Alex leaves her apartment
Requirement Link ——
Contingent Link ----

Figure 3-2: Sam and Alex go to the movies, but Sam's phone needs to recharge for 40 minutes to let her call Alex. Figure for Example 3.2.

> 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. They need to enter the theater at the same time because the movie is sold out, and they want to get seats together. Sam's phone is **deeply discharged** and needs to be recharged before calling Alex. When Sam arrives at the museum, she will leave her phone to charge at the museum front desk for **40 minutes**, before calling Alex.

Example 3.2 is not controllable. Unlike the preceding example, if it takes Sam 40 minutes to drive to the museum, she will only be able to call Alex at 8:20pm, which is five minutes after the movie starts. To account for this possible lack of information, Alex could reason that she must leave the house by 8pm if she has not yet received a call in order to guarantee that she arrives to the movie on time. At the other extreme, if Sam arrives at the movie theater as early as 7:21pm, she will be finished with the Museum of Bad Art by 8:06pm. Since it takes 40 minutes to charge her phone, she will not be able to call Alex before 8pm, which does not leave Alex enough lead time to get to the theater at the same time. In this situation, in order for the two to arrive at the movie theater at the same time, Sam must spend 54 minutes at the Museum of Bad Art, violating her preexisting constraint.

The only difference between the two examples is the amount of time it takes for Sam's phone to recharge, or to put it differently, the amount of time that elapses before Alex can find out when Sam arrived. Even though the set of actions taken by all agents is the same, communication has an impact on the feasibility of the scheduling problem.

Existing controllability models do not appropriately model the difference between these two examples. Neither problem instance can be pre-scheduled using strong controllability, as no single schedule will work for Sam across all scenarios. But as we showed, it is possible to construct a valid schedule for Example 3.1 on the fly. In contrast, when we use dynamic controllability, we discard all the difficulties of dealing with limited communication and communication delay, and our model tells us that both problem instances are controllable, when in fact Example 3.2 is not. 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 communication delays that make coordination difficult in Example 3.2. Being able to model delay in communication motivates our decision to introduce a new form of controllability.

Definitions

To introduce delay controllability more rigorously, 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.

We want our delay function to be highly expressive. For example, it is important that we allow agents to have different delays in their communications and to give agents the option of choosing the delay for each of their actions. We should also allow for certain events to be marked unobservable, precluding anyone from acting on that information. Finally, our model should be sufficient to support the reasoning used in evaluating Examples 3.1 and 3.2, to explain why the first is controllable and the other is not.

Definition 3.1. Delay Function
A delay function, $\gamma : X_c \rightarrow \mathbb{R}^+ \cup \{\infty\}$, takes a contingent event as input and outputs the amount of time that will pass between when it occurs and when its value is observed.

This model of delay is powerful. For example, it allows us to model communication outages as they pertain to specific events; for example, in an STNU with contingent events x_1 through x_6, we can capture the idea that we receive communication about all events except x_1, x_2, and x_3 by constructing a delay function with $\gamma(x_1) = \gamma(x_2) = \gamma(x_3) = \infty$ and for all other x, $\gamma(x) = 0$. Similarly, different events can have different delay patterns. We can easily let $\gamma(x_4) = 12$, $\gamma(x_5) = 0.01$, and $\gamma(x_6) = 0$ in the same model, to represent the variability in communication across different events and agents.

This model of delay, however, cannot express delay based on the moment in time in which the event happens; for example, that communication is more delayed outside of normal working hours. In other words, we say that our delay function γ is stationary. Modeling this type of variability has the potential to introduce non-linear and conditional constraints that would significantly affect our ability to solve the scheduling problem quickly. Our model does not allows us to approximate schedules that involve non-stationary delays, and exploring this in detail is outside the scope of this thesis.

This definition of a delay function also allows us to add detail to STNUs. In particular, they allow us to describe the specific situations highlighted in Examples 2.1, 3.1, and 3.2, and their differences. The STNUs for the two examples with a drained cell phone are nearly equivalent. They only differ in the amount of time that passes before Alex learns the specific timing of Sam's commute. In both examples, there is only one contingent event, $X_e = \{B\}$; in Example 3.1, we use $\gamma(B) = 5$ to represent that Alex learns of Sam's arrival at the movie theater 5 minutes after it happens, and in Example 3.2, we use $\gamma(B) = 40$ to represent that Alex learns of Sam's arrival after 40 minutes.

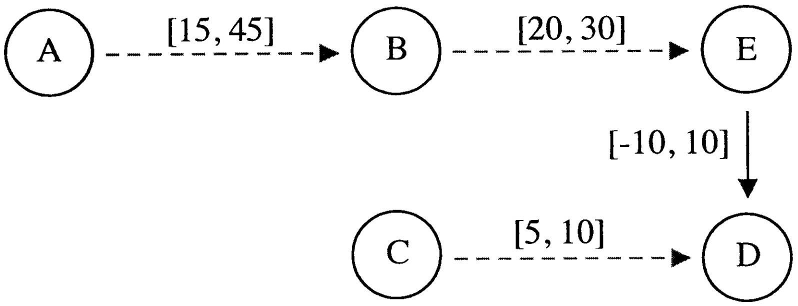
Given a delay function that characterizes communication of events in an STNU, we want to be able to answer the question of whether it is possible to construct a feasible schedule given these communication constraints. In particular, we want to be able to construct a schedule on the fly that is flexible enough to adapt to new observations, but does not require all information eventually made available. *Delay controllability* answers these questions. Specifically, we say that an STNU is delay controllable with respect to some delay function γ if it is always possible to construct a satisfying schedule to future events given the events that have already been observed.

Definition 3.2. Delay Controllability
An STNU S is delay controllable with respect to some delay function γ if and only if for any set of contingent link durations that respect S's contingent constraints, it is possible to construct a satisfying schedule on the fly, assuming that each contingent event x_c is observed after an additional $\gamma(x_c)$ units of time have passed.

Under the model of delay controllability, we can easily model strong and dynamic controllability. An STNU S is strongly controllable if and only if it is delay controllable with respect to delay function $\gamma(x_c) = \infty$, for all x_c. We also know that an STNU S is dynamically controllable if and only if it is delay controllable with respect to delay function $\gamma(x_c) = 0$, for all x_c. Beyond strong and dynamic controllability, delay controllability provides us the ability to reason about scenarios like the ones we highlighted in Examples 3.1 and 3.2. In the rest of this thesis, we build on this simple abstraction to check delay controllability and to efficiently schedule systems that are delay controllable.

3.2.2 Managing Communication Costs

Next, we frame the problem of deciding when to communicate. Delay controllability as presented goes a long way towards satisfying our ideal set of requirements, but it assumes that there are predetermined moments in time when communication will happen (i.e. Sam will call Alex exactly 5 minutes after she arrives at the museum). In reality, agents often have flexibility in choosing when to communicate the results of their actions (i.e. Sam can choose to wait until she has done a lap of the museum



A: 8am
B: Sam starts driving to the coffee shop
C: Alex starts walking to the coffee shop
D: Alex arrives at the coffee shop
E: Sam arrives at the coffee shop

Figure 3-3: Sam and Alex plan to meet for coffee. Figure for Example 3.3.

before calling Alex). Further, communicating more actively or more frequently can come at an increased cost, whether in terms of bandwidth, attention, or something else entirely. Agents then may choose to schedule their communication in a way that minimizes this cost while guaranteeing controllability.

Delay controllability can only answer the yes-or-no question of whether a system is controllable, but often we want to understand how long we can delay communication and still guarantee the overall success of the schedule. When there are costs associated with communicating, we then want to know what the optimal (or at least some good) set of delays is that still guarantees success. We provide algorithms for solving this problem in Chapter 5; here, we start by introducing the *Communication Cost Minimization Problem* (CCMP) and showing the types of scenarios we aim to model with it.

Examples

Consider the following example of a situation where it is important to decide when to communicate, in order to guarantee plan feasibility:

Example 3.3. Sam and Alex plan to meet for coffee. See Figure 3-3.

> Sam and Alex have decided to meet for coffee. Alex lives a short 5-10 minute walk from the coffee shop and leaves it up to Sam to decide exactly when they will meet. Sam wants to leave home some time between 8:15am and 8:45am, and it takes 20-30 minutes to drive to the coffee shop. Neither wants to spend more than 10 minutes waiting for the other to arrive.

Example 3.3 is not strongly controllable. Without any communication, Sam could show up anywhere between 8:35am and 9:15am, and there is no way to guarantee that neither will be waiting more than ten minutes for the other to arrive. It is, however, dynamically controllable; when Alex observes Sam's activities, Alex can always be guaranteed to arrive at the coffee shop within ten minutes of Sam's arrival. We thus know that some form of communication between the agents is necessary to guarantee that all temporal constraints will be satisfied. The act of picking a communication policy that guarantees success is equivalent to picking a delay function γ such that the original STNU is delay controllable with respect to γ.

Because the scenario is dynamically controllable, we know there is at least some choice of delay function γ that guarantees successful execution of the schedule (trivially $\gamma(x_c) = 0$ for all x_c), and depending on the semantics of the problem, we may preferentially choose one over others. In this specific example, we may argue that driving is a mentally demanding activity and that, all else equal, it would be preferable for Sam not to communicate from the car. Given this preference, we can construct a feasible plan whose communication policy (or delay function) respects the stated preference. As long as Sam communicates with Alex the instant she arrives, Alex can leave her apartment and ensure that Sam spends no more than ten minutes waiting and can avoid calling while driving.

Below we introduce the problem of finding such an optimal delay function. It is important to realize that communication during plan execution may deviate from the predetermined optimum due to errors in execution or uncertainty in the underlying model. Another problem we introduce, and in Chapter 5 provide algorithms for, is the problem of re-planning under communication changes.

To illustrate the effects of uncertainty during execution, consider Example 3.3

once again. While we determined that the optimum strategy was for Sam to communicate with Alex immediately after arriving, during execution Sam may deviate from the optimal communication strategy, calling Alex from the car to tell her when she originally left. Given this communication, it is now superfluous for Sam to call Alex again after she arrives. In this case, given that Alex knows when Sam left home, she can plan to leave her own home 15 minutes later. This guarantees that Alex will arrive between 20 and 25 minutes after Sam leaves home and that Sam will now spend no more than 5 minutes waiting for Alex. Had we maintained our original communication strategy, Sam would have had to make two calls; instead, we can minimize the amount of communication that happens by adjusting our strategy in response to deviations during execution.

Definitions

A few important definitions will assist with more rigorous definitions of the problems identified in this subsection.

Definition 3.3. Delay-Cost Function
A *delay-cost function* C takes in a delay function γ and outputs a real-valued cost.

Intuitively the purpose of a delay-cost function is to quantify the burden of communication in a multi-agent scenario. If communicating is easy and free, then the delay-cost function is $C(\gamma) = 0$, for all delay functions γ. More natural delay-cost functions might include, for example, bandwidth (perhaps approximated by counting the number of times that any communication happens).

With this notion of what constitutes a delay-cost function, we can introduce the Communication Cost Minimization Problem (CCMP).

Definition 3.4. Communication Cost Minimization Problem (CCMP)
The Communication Cost Minimization Problem (CCMP) takes in a tuple $\langle S, C \rangle$, where S is an STNU and C is a delay-cost function, and outputs the minimum cost delay function γ that guarantees that S is delay controllable.

In constructing algorithms to solve CCMPs, we often consider only *admissible* delay-cost functions. We say that a delay-cost function is admissible if it enforces that a delay function with monotonically larger delays is no more costly than one with smaller delays. Our decision to restrict the input of CCMPs to admissible delay-cost functions is not restrictive in practice, as most models of communication operate under the assumption that it is more difficult or costly to be proactive about communication than to delay communication.

Definition 3.5. An *admissible* delay-cost function is a delay-cost function C that is component-wise monotonically decreasing. That is, given two delay functions γ_1 and γ_2, such that for all contingent events x_c, $\gamma_1(x_c) \leq \gamma_2(x_c)$, we have that $C(\gamma_1) \geq C(\gamma_2)$.

For temporal scheduling, we know that we can improve robustness through a scheduling policy that dynamically adapts and schedules events on the fly. We can achieve similar improvements by adapting communication delays on the fly to account for uncertainty during execution. We present an approach for this in Chapter 5.

3.2.3 Variable-Delay Communication

Our final modeling contribution introduces uncertainty in communication. The model of delay controllability presented to this point assumes that when communication happens, the exact timing of past events are learned. In other words, this model assumes that communication around events, though delayed, always comes with perfect precision and complete accuracy. However, this delay may not be precisely known due to approximations by the model or to physical uncertainty in communication channels. *Variable-delay controllability* and *variable-delay functions* provide a way to model that ambiguity and noise in communication events, so that they eventually get evaluated in an algorithmic context.

Examples

To illustrate the impact of communication uncertainty on delay controllability, consider the following example:

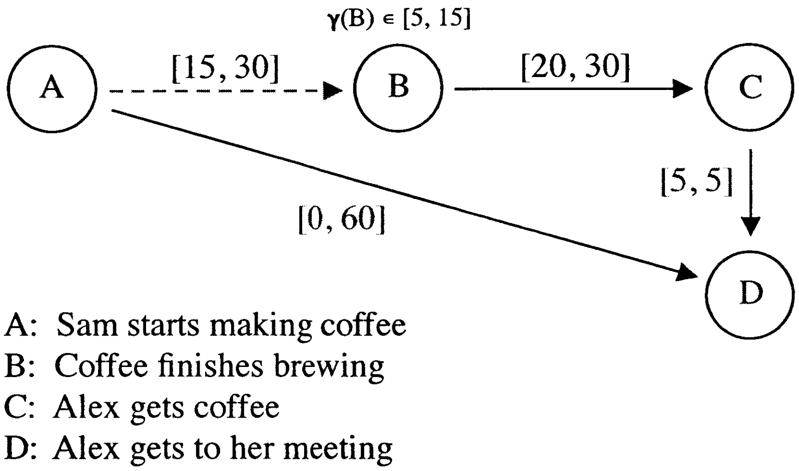

γ(B) ∈ [5, 15]
A
[15, 30]
B
[20, 30]
C
[5, 5]
[0, 60]
D
A: Sam starts making coffee
B: Coffee finishes brewing
C: Alex gets coffee
D: Alex gets to her meeting

Figure 3-4: Sam brews coffee, and Alex wants to have some after it has cooled down. Figure for Example 3.4.

Example 3.4. Alex wants some coffee that Sam brewed. See Figure 3-4.

> At 8am every morning, Sam brews a large pot of coffee for the office. The machine Sam uses is quite old and temperamental, so it can take anywhere from 15 to 30 minutes to get the coffee ready. The coffee machine is several rooms over, but Sam sends an email to the group that the coffee is ready after he finishes his cup; drinking a cup of coffee takes Sam somewhere between 5 and 15 minutes. Alex wants to get the coffee after it has cooled a bit and finds the temperature to be optimal between 20 and 30 minutes after the coffee is ready. It takes Alex 5 minutes to finish her coffee, and once she is done, she will go to a client meeting that starts at 9am.

Unlike the previous examples we presented, communication in this scenario does not offer absolute certainty about when to act. The only communication that Alex receives from Sam is in the form of an email that comes at some indeterminate amount of time after when the coffee is brewed.

Whenever Alex receives the email before 8:40am, it is simple for her to come up with an execution policy. Because she knows the email arrives between 5 and 15 minutes after the coffee is brewed, if she waits 15 minutes from when the email arrives, she has a guarantee that she can get her coffee between 20 and 30 minutes after it is brewed. The challenge is to address the situation where the email arrives after 8:40am. In that instance, Alex cannot wait 15 minutes to get her coffee, as that will make her late for her meeting.

To address the situation, we reason about the underlying distribution associated with the time it takes to brew the coffee directly and for the email to be sent. Specifically, Alex knows that it takes at most 15 minutes for the email to arrive. If an email has not arrived by 8:40am, the coffee must have taken at least 25 minutes to brew. As a result, if Alex gets coffee at 8:55am, she still has a guarantee that the coffee has been sitting out for somewhere between 25 and 30 minutes. This allows her to get coffee that is optimally fresh and still attend her meeting on time. Variable-delay controllability formalizes this reasoning process.

Our ability to reason about this example relies on the fact that there are strict bounds on the window of communication associated with an event; for example, that we know with certainty that Sam's email cannot come after 25 minutes.

In reality, there are many reasons why this model might be inaccurate and that Sam's email could arrive after 25 minutes. Sam might get caught up in conversation with someone else causing an unexpected delay. Or, the department's email infrastructure might be under tremendous load causing the email to arrive in Alex's inbox well after Sam sent it. Or, most simply of all, Sam might forget to send the email. Sam's communication tendencies are better represented by a probability distribution than a simple set interval.

Given the true nature of this uncertainty, we may also determine that we cannot put an upper-bound on the delay in communication, and unfortunately, this would make our previous approach impossible. But the reality of this situation is that these events are highly unlikely. Given the distribution of all possible times that Alex could read Sam's email, it is impossible to guarantee 100% success, but in very few cases is 100% guaranteed success necessary. An important feature of our formalism should be that it is able to identify temporal plans that have extremely high chances of success or, to put it differently, that it is able to guarantee that the probability of failure stays below a certain predefined threshold.

Definitions

To appropriately model the examples we introduce above, we must augment our models of delay controllability to include communication that is itself uncertain. To represent uncertainty in our models, we introduce the concepts of *variable-delay function* and *variable-delay controllability.*

Definition 3.6. Variable-Delay Function

A variable-delay function, $\bar{\gamma} : X_c \rightarrow (\mathbb{R}^+ \cup \{\infty\}) \times (\mathbb{R}^+ \cup \{\infty\})$, takes as input a contingent event and outputs an interval $[a, b]$. The range bounds the time that may pass after the assignment of a value to the contingent event, before that value is known to be assigned.

Importantly, this model does not assume that the event's assigned value is known, but instead just that the event is known to have happened. By convention, we use $\bar{\gamma}^-(x_c)$ and $\bar{\gamma}^+(x_c)$ to represent the lower-bound and upper-bound in observation of contingent event x_c, respectively.

Like the delay function for fixed-delay controllability, the variable-delay function independently describes the delays in observation for each contingent event. We apply a similar approach in extending the delay function to a variable-delay function to introduce the definition of variable-delay controllability from delay controllability.

Definition 3.7. Variable-Delay Controllability

An STNU S is variable-delay controllable with respect to a variable-delay function $\bar{\gamma}$ if and only if for any set of contingent link durations that respect S's contingent constraints, it is possible to construct a satisfying schedule on the fly, assuming that each contingent event x_c is observed at some interval $[\bar{\gamma}^-(x_c), \bar{\gamma}^+(x_c)]$ after x_c actually occurs.

Variable-delay controllability is a generalization of fixed-delay controllability. For any fixed-delay function γ, we can produce a corresponding variable-delay function $\bar{\gamma}$ where $\bar{\gamma}^+(x_c) = \bar{\gamma}^-(x_c) = \gamma(x_c)$.

When our models of uncertainty in communication are better represented by distributions and in particular when those distributions are unbounded, we often know that there cannot be a complete guarantee of success. Instead, we use *chance-constrained variable-delay controllability* to determine whether an STNU is controllable assuming we admit a small risk of failure.

To define chance-constrained variable-delay controllability, we first have to update our definition of what constitutes a delay function.

Definition 3.8. Chance-Constrained Variable-Delay Function

A chance-constrained variable-delay function, $\bar{\gamma} : X_c \to \mathbb{P}^+$, takes in a contingent event and outputs a probability distribution function that is defined over the range $\mathbb{R} \cup \{\infty\}$, representing the likelihood that communication about event x_c happens after some amount of time has passed.

We say that a particular grounding of a variable-delay function is a fixed delay function γ with all $\gamma(x_c) \in \bar{\gamma}(x_c)$.

We now use our chance-constrained variable-delay function to define chance-constrained variable-delay controllability.

Definition 3.9. Chance-Constrained Variable-Delay Controllability

For execution strategy s, we define an indicator function c_s such that $c_s(\gamma) = 0$ if strategy s satisfies all constraints of a given STNU during execution for some $\gamma \in \bar{\gamma}$ and 1 otherwise. We say that STNU is chance-constrained variable-delay controllable with respect to delay function $\bar{\gamma}$ and tolerated risk $\Delta \in [0, 1]$ if there exists an execution strategy s such that:

$$\int_{\gamma \in \bar{\gamma}} p(\gamma) c_s(\gamma) \leq \Delta$$

The modeling problem of specifying an optimal risk-bound for the chance-constrained variant of the problem is in general difficult, but in Chapter 6 we establish a series of results for the set-bounded version of the problem and show how to use ideas from the set-bounded variants to verify the chance-constrained controllability of analogous problems.

3.3 Related Work

The areas of focus suggested by this chapter's desiderata have been tackled in certain ways by many others over the years. In particular, there are three strong candidate frameworks for temporal reasoning in a multi-agent context that are worth highlighting to illustrate how this work builds upon the strengths of work that has come before. The first, ϵ-dynamic controllability, operates on many temporal networks, including the STNU, and introduces delay in the form of a reaction time that is persistent and uniform across all contingent events. The second involves Partially Observable STNUs (POSTNUs), which augment the STNU model by individually marking contingent events as observable or unobservable. The third, Multi-agent Simple Temporal Networks with Uncertainty (MaSTNUs), goes even further and models observability of events distinctly for each individual agent.

3.3.1 ϵ-dynamic controllability

The first framework, ϵ-dynamic controllability, puts forth a new type of controllability for evaluating temporal networks with uncertainty. Most dynamic controllability models assume that the scheduling agent has an instantaneous reaction time; in other words, they can schedule executable events to immediately co-occur with other contingent events. In practice, this co-occurrence is infeasible. Even the most high precision robotic systems have a sense-react loop where some computation occurs in between when an event happens and when the system acknowledges that event and issues a response.

ϵ-dynamic consistency is a way of validating whether an execution strategy exists for a network when agents have non-instantaneous reaction times in Conditional Simple Temporal Networks with Uncertainty (CSTNUs) [19, 29]. In this model, the reaction time for the scheduler to observe and react to contingent events is parameterized by the value ϵ. Though this concept was defined over CSTNUs, it is quite natural to extend the same reasoning to STNUs and to discuss ϵ-dynamic controllability.

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 not able to deal with varying delays, due to either delayed communication of observations or variability in processing time; at its core, ϵ-dynamic controllability is equivalent to delay controllability with respect to $\gamma(x_c) = \epsilon$, 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 3.1).

3.3.2 POSTNUs

One of the strengths of the delay controllability model is its ability to blend notions of strong and dynamic controllability. This technique was first seen in greater depth in the context of POSTNUs [36]. In an STNU, all contingent events are either instantaneously observable under a dynamic controllability model or entirely unobservable under a strong controllability one. In POSTNUs, contingent events can be marked observable and unobservable. To say that a POSTNU is dynamically controllable is the same as saying that it is possible to construct a schedule on the fly that respects all constraints if the scheduler only receives information about observable contingent events. While POSTNUs are more expressive than STNUs with delay, it is not clear that they are more useful in practice; the sub-class of POSTNUs that can be checked efficiently and accurately can all be expressed directly as STNUs with delay. The advantage of using STNUs with delay directly is that we have a guarantee that the outputted model can be checked efficiently.

In Figure 3-5, we provide an example to illustrate how one might model delay controllability using POSTNUs. However, unlike with delay controllability checking, we do not have a guarantee that POSTNU dynamic controllability can be evaluated in polynomial time. Currently, the best known POSTNU checker is only sound and complete for networks that lack *chained contingencies* [10], where chained contingencies are defined as series of two or more successive contingent constraints $A \Rightarrow B \Rightarrow C$ where each middle node B is also involved in another contingent or requirement constraint (see Figure 3-5b).

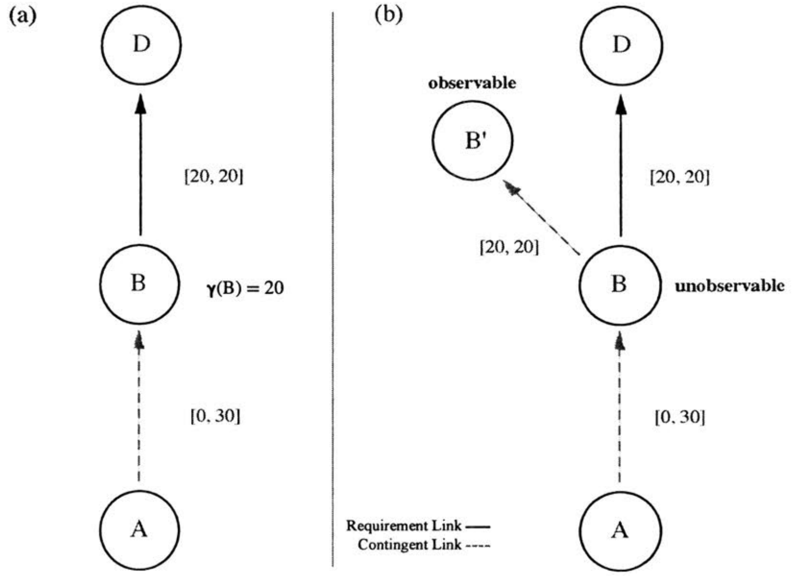


Figure 3-5: (a) An STNU with a contingent constraint that has a certain delay. (b) One possible way of rewriting the STNU as an equivalent POSTNU. This particular POSTNU exhibits a chained contingency, as B is a contingent event that starts a contingent constraint and is connected to B' via a contingent constraint.

However, all POSTNUs that can be checked efficiently (i.e. those that are free of chained contingencies) can be modeled directly into STNUs with delay. To transform a POSTNU P into STNU S with delay function γ, we only need to focus on transforming the contingent constraints.

We start by dividing our POSTNU contingent constraints based on whether they are observable or not and then further subdivide them based on whether their terminal contingent events are immediately followed by other contingent constraints. For the unobservable contingent constraints that have no successor contingent constraints, we keep the original contingent constraint and ensure that its terminal contingent event, x_c, respects $\gamma(x_c) = 0$. For an unobservable contingent constraint $A \overset{[u,v]}{\Longrightarrow} C$ that is followed by a successor contingent constraint $C \overset{[w,z]}{\Longrightarrow} D$, we replace the two constraints in our POSTNU with a new $A \overset{[u+w,v+z]}{\Longrightarrow} D$ whose observability is the same as the second constraint. 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 constraint does not affect the semantics of the network. All that remains to complete the transformation is to handle observable contingent constraints. For every observable contingent constraint $A \xRightarrow{[u,v]} C$ in the original POSTNU, we replace it with a contingent constraint $A \xRightarrow{[u,v]} C'$ and a requirement constraint $C' \xrightarrow{[0,0]} C$ in our STNU, where $\gamma(C') = 0$. Since C was observable in the POSTNU, we can simulate executing an event immediately as it is observed, meaning that any contingent constraints that follow $A \xRightarrow{[u,v]} C$ would now start at an executable event. Thus, our delay controllability framework is sufficiently capable of expressing all POSTNUs that can be efficiently checked for controllability using today's tractable POSTNU algorithms.

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. In contrast, all STNUs with delay are guaranteed to be evaluated efficiently, and to the extent that scenarios can be modeled directly as STNUs with delay, this provides an advantage to our model.

Consider the transformation in Figure 3-5. The transformation converts an STNU with delay function into a POSTNU that has a chained contingency. However, 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 since when we use POSTNUs, an errant choice when modeling can make evaluation intractable. This distinction makes delay controllability an important concept to use and build upon at least until more complete algorithms are discovered for POSTNUs.

In fact, it is worth noting that the work presented in this thesis for variable-delay controllability can be leveraged to construct an improved algorithm for POSTNUs. The model proposed by variable-delay controllability looks exactly like the chained contingency shown in Figure 3-5b; the main difference is that we represent the contingent link between B and B' with our variable-delay function $\bar{\gamma}$. Hence, the algorithm we eventually present in Chapter 6 for variable-delay controllability can be used to both solve POSTNUs without chained contingencies, as well as those with chained contingencies that can be expressed as variable-delay functions. It is for this reason that we believe that delay controllability is a useful framework for modelers and represents a solid grounding for future work.

3.3.3 MaSTNUs

Third, we consider the MaSTNU [15]. The MaSTNU behaves exactly as an STNU does but assigns each event to a set of agents to determine who can observe that event. We say that an MaSTNU is dynamically controllable if each agent can plan across the events that they themselves can observe, such that all constraints are satisfied.

MaSTNUs offer a significant degree of expressiveness above and beyond what the STNU or POSTNU can offer. However, this expressiveness comes at a cost. While efficient algorithms exist for solving some subset of MaSTNUs [15], no efficient algorithms are known to exist that correctly determine the controllability of arbitrary MaSTNUs. While work in Appendix A provides an upper-bound (NEXP) on the runtime through a disjunctive variant of MaSTNUs, it remains an open question whether MaSTNUs themselves are similarly intractable.

One of our goals in providing a set of modeling tools is to ensure that there exist efficient algorithms that can be used across those models. While it is possible that, in the future, efficient algorithms are found for MaSTNUs, the modeling framework we put forth based on delay controllability comes with a set of algorithms that are efficient enough to use in practice.

3.4 Additional Examples

To conclude this chapter, we provide two grounded examples of multi-agent coordination tasks and demonstrate how to model these problems given our framework. The work presented in this thesis provides a series of algorithms for evaluating the controllability and execution of temporal networks under limited communication, but the true impact of this work is felt by those interested in modeling temporal problems.

We believe that the work presented here can be used to build larger, more capable systems that have to deal with multiple agents, and believe that our work can be used flexibly across many situations that deal with communication.

3.4.1 Coordinating Autonomous Underwater Vehicles

The autonomous vehicle problem described in Example 1.1 is a classic example of multi-agent coordination under limited communication. What makes the problem challenging is that the AUVs need a guarantee that they are operating continuously despite being in distinct regions, and despite being unable to communicate directly. They must use either a central ship or shore operations as intermediaries for their communication and can communicate only when surfaced.

The problem of constructing an appropriate plan reduces to one of finding an appropriate set of times for the AUVs to surface in order to guarantee sufficient communication. In other words, we want a way to evaluate whether, given a set of planned surfacings, there is enough information to prevent AUVs from operating in the same location. Our approach for doing so is to construct a delay function γ for each vehicle v representing the amount of delay between when some other vehicle w acts and when v learns the result of w's action.

The simplest way to approach this problem is to introduce a minimum rate at which each vehicle surfaces. For examples, v could surface every 9 hours, and w could surface every 6 hours. This gives us at worst a 9 hour delay between when w reports the results of its actions back to shore operations and when v learns the results (since w's report may have arrived immediately after v's previous surfacing). There is at most a 6 hour delay between when w takes an action and when it reports that action. Hence, there is at most a 15 hour delay in observation between vehicles v and w.

In many instances, this amount of delay is sufficient to ensure proper operations if the AUVs are in general far away from each other and operating over large distances and timescales. In the event that this approach is insufficient, is possible to adopt more complex surfacing strategies to improve the level of coordination between vehicles. So long as it is possible to calculate the maximum amount of time that can elapse between an event happening and a vehicle learning about it, it is possible to use delay controllability to evaluate the feasibility of a set of planned surfacings.

3.4.2 Robotics

Next, we consider how the techniques presented in this thesis can be used to improve the operation of an in-home robotic assistant.

Quantifying Success

One goal of an effective in-home robotic assistant would be to anticipate the needs of their human partner and to be proactive in the execution of household chores, which can include things like meal prep, cleaning, and finding lost items scattered throughout the home. In order to determine what the needs of its human counterpart are, most robots are likely to have an on-board state estimation system that provides estimates about the state of the world. From these estimates, the robot can then determine how best to support its human counterpart.

However, since estimates are probabilistic at best, we want to quantify the likelihood that a robotic agent can execute a given plan successfully. To do so, we construct a CCMP and use it to bound the overall probability of success.

First, we must consider the space of available delay functions. For each contingent event, the robot can either infer that it occurred or fail to make the inference, meaning that for each x_c, either $\gamma(x_c) = 0$ or $\gamma(x_c) = \infty$. We assume that some a priori probability distribution is known that characterizes how likely the robot is to infer that the event actually occurred.

Given this setup, it is relatively straightforward to model this kind of problem, and in fact, we can use the same generalized technique for all possible types of delay functions. If we let d be a function mapping contingent events to proposed bounds on their delay, we can use a cost function $C(\gamma) = 1 - P\left(\bigwedge_{x_c} [\gamma(x_c) \leq d(x_c)]\right)$. If we find an optimal cost solution, our probability of success is bounded below by $1 - C(\gamma)$.

Amazingly, our approach does not make any real assumptions about the probability distribution itself, other than the fact that it is computable.

With this cost function, we can then apply the algorithms from Chapter 5 to determine the minimum cost that still guarantees successful execution which in turn gives us a lower-bound on our solution.

Asking for Help

We now have the ability to use CCMPs to quantify the likelihood of success of an in-home robotic agent. But the previous problem assumed that the robot acted passively, waiting for information to arrive about the world. In reality however, an effective robotic agent must also be able to ask for help, and when it finds itself in a situation where it is uncertain about what is happening, asking a human counterpart for more information can help repair situations that were otherwise marked as failures in the previous characterization. While asking in all instances is a way to guarantee success, a large number of queries from the robotic agent is likely to be undesirable from the perspective of its human counterpart.

Again, we construct a CCMP for our situation, but in this instance, we use a cost function that counts the number of times the robot interrupts its human counterpart. In other words, we let $C(\gamma) = |\{\gamma(x_c) \neq \infty\}|$. When we solve this CCMP, we then get a contingency plan for our robotic agent. For any solution γ to the CCMP, the robot now modifies its policy; for each contingent event x_c, if the robot does not have enough certainty to determine that x_c has occurred within $\gamma(x_c)$ time of when it may have earliest occurred, then it asks the robotic counterpart for help. By adopting this policy, the estimated number of interruptions is now upper-bounded by the result returned by the CCMP and in fact can be improved in practice during execution.

In Chapter 5, we show how we can improve this approach even further. The delay function γ representing the solution to our CCMP operates pessimistically. It assumes that no contingent event x_c is observed sooner than $\gamma(x_c)$ after it occurs. In reality, however, there may be several instances in which it is observed sooner. In such cases, the adaptive algorithms from Chapter 5 can be leveraged to further refine the optimal choice of γ to guarantee success while, in this case, minimizing further interruptions.

Chapter 4

Delay Controllability

Consider the difficulty of scheduling events in a scenario where multiple agents have to coordinate with each other. From the perspective of any one agent, the possible times of actions of other agents may be bounded, but the actual time taken for any given action can be highly uncertain; planning under this kind of uncertainty can present a real challenge. If all agents are in constant communication with one another, creating a schedule under uncertainty reduces to a known problem, that of determining dynamic controllability. However, the assumption of constant and instantaneous communication is not a realistic one. Delay controllability offers a way to model these types of problems, and in this chapter, we introduce the algorithms and corresponding proof theory that illustrate how to check delay controllability efficiently. In this chapter and Chapter 6, we introduce a model for dynamic scheduling under uncertainty that factors in communication constraints. In this chapter, we consider that communication is delay but that delay is deterministic. In Chapter 6, we generalize this model to the case where communication itself is uncertain.

Existing controllability approaches can be less than ideal for modeling these types of multi-agent problems. Agents might only relay the relevant information and may only offer it after some indeterminate delay or perhaps not at all; hence dynamic controllability tends to be overly optimistic and not robust. Strong controllability scheduling techniques give valid solutions by using pre-scheduling to remove the need to communicate. However, too often the strong controllability approach is overly conservative, as it assumes that external uncertainty is never reduced along the way through observations. Delay controllability provides a way to create robust schedules for realistic problems with communication outages and delays without being overly conservative.

One of the strengths of delay controllability as a model is that it generalizes the principles of dynamic and strong controllability, allowing us to build on the shoulders of previously established research. We apply familiar concepts while retaining the runtime complexity and efficiency of the best of these algorithms. In this chapter in particular, we show how to infer new constraints on STNUs and check controllability by extending and generalizing a set of constraint derivation rules provided by [40]. This enables us to adapt a highly efficient dynamic controllability checking algorithm to handle delay controllability [38]. We also demonstrate correctness by modifying a known dynamic controllability execution strategy and its proof of correctness [28].

Beyond building a polynomial-time delay controllability checker, there are many interesting bodies of work that could be extended by incorporating 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. It uses dynamic controllability to determine appropriate execution strategies [18] and could similarly admit delays in the observation of these conditions. For more specialized use cases, incremental dynamic controllability checks show improvements over non-incremental solutions [6, 41, 42, 47, 48], and the same approaches likely transfer over to provide incremental variants of delay controllability checking. 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.

This chapter provides the formal foundations for solving delay controllability problems as was defined in Chapter 3. We provide an efficient sound and complete algorithm for checking delay controllability, which demonstrates that our generalization allows us to capture and describe many scenarios with restricted communication, without sacrificing 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 instances of a more general idea.

We validate this work 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 yields execution policies that are either inaccurate or overly conservative. We also demonstrate that delay controllability is fast enough in practice for use as a subroutine in large planning and execution systems.

4.1 Approach

In this chapter, we present a series of algorithms for determining whether a particular STNU is *delay controllable* with respect to a particular delay function γ (formal definitions can be found in Chapter 3). Our approach for producing such an algorithm proceeds as follows.

We start by showing how to perform propagation on the constraints encoded in an STNU in order to generate new constraints. Constraint propagation is performed in service of finding a *semi-reducible negative cycle*, and we correspondingly show that an STNU is delay controllable if and only if the STNU's constraints do not entail the presence of a semi-reducible negative cycle. We conclude by presenting an algorithm, DELAYDIJKSTRA, and showing that the algorithm is able to determine whether it is possible to generate semi-reducible negative cycles in an STNU in $O(n^3)$ time.

4.2 Constraint Propagation

Much as we did with STNs, to evaluate STNU delay controllability we use graphical formalisms to represent and reason about STNUs and their constraints directly (see Figure 4-1a). In these graphs, we use solid arrows to represent requirement constraints and dashed-line arrows to represent contingent constraints. When we evaluate STNU delay controllability, we similarly use an extended version of the STN distance graph,

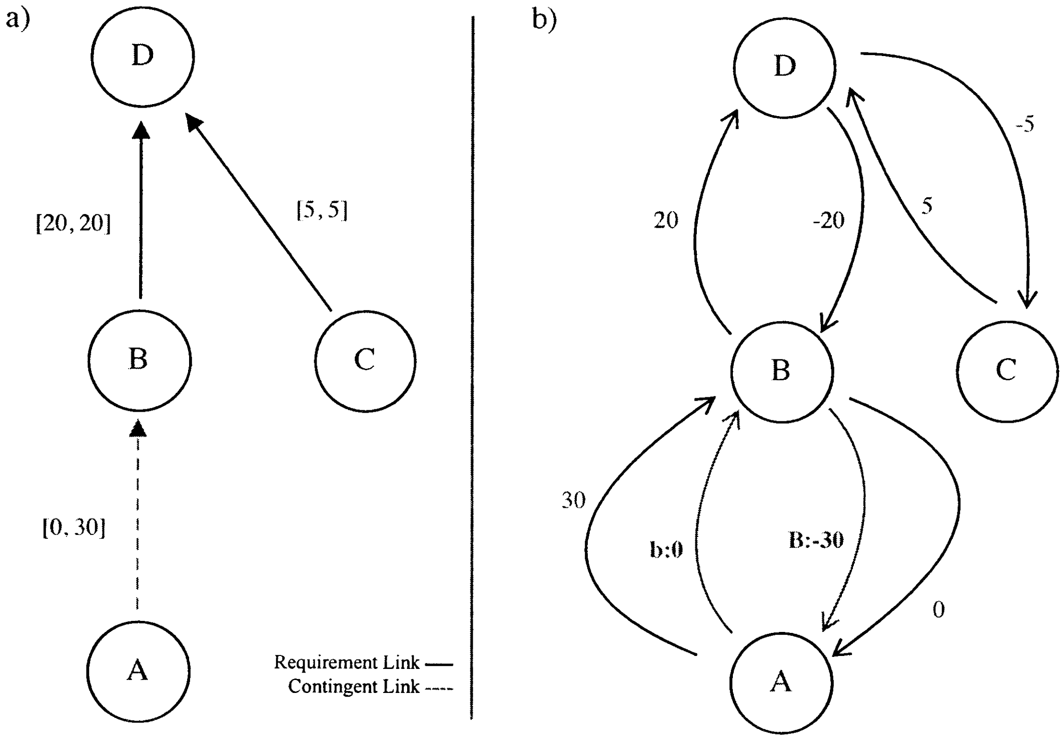


Figure 4-1: (a) A graphically represented STNU. (b) The same STNU represented using its labeled distance graph formulation.

called the *labeled distance graph* [37] (see Figure 4-1b). We start by describing the semantics of labeled edges and then describe how to produce a labeled distance graph from an STNU.

Semantically, the edges of a distance graph represent constraints. An edge of the form $A \xrightarrow{u} C$ implies that $C - A \leq u$. The purpose of labeled edges in an STNU's labeled distance graph is to represent conditional constraints. Because an STNU involves events that are outside the scheduler's control, it becomes necessary to reason conditionally about scheduling requirements that selectively apply, depending on the durations of contingent constraints; it is possible that if a contingent constraint takes on its minimum possible duration, the scheduler responds in one way, but if it takes on its maximum possible duration, the scheduler may respond entirely differently.

In the case of labeled edges, we say that a *lower-case* edge of the form $A \xrightarrow{c:x} C$ enforces the constraint "whenever the duration of the contingent constraint ending

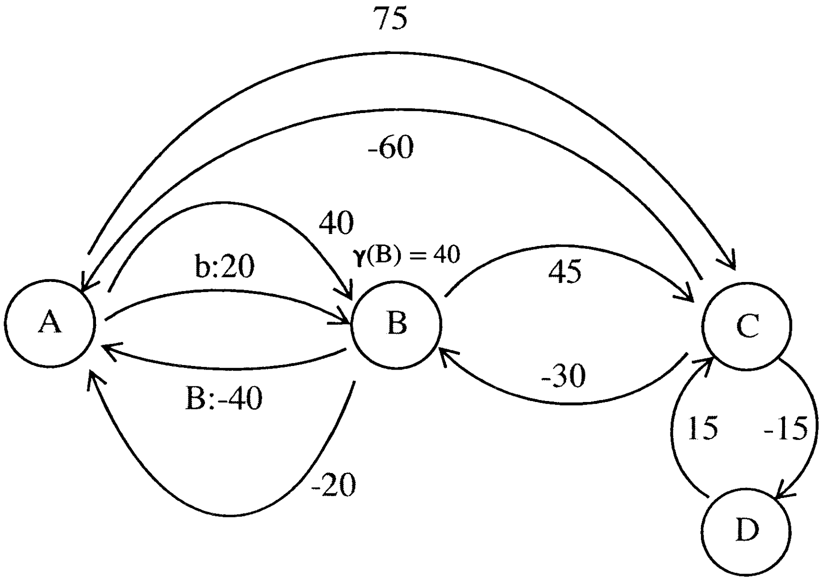


Figure 4-2: A recreation of Example 3.2 in labeled distance graph form.

at C were to take on its lowest possible value, $C - A \leq x$". Similarly, an *upper-case* edge of the form $B \xrightarrow{C:x} A$ enforces "$A - B \leq x$ whenever the duration of the contingent constraint ending at C takes on its maximal possible value". With our representation of constraints as edges, our algorithm derives implied path constraints (both conditional and unconditional) by considering paths through the graph.

To generate the labeled distance graph, we first treat the STNU as an STN, converting contingent constraints to requirement constraints, and generate the derived STN's distance graph. We then add additional labeled edges to the graph for each contingent constraint. Given a contingent constraint of the form $A \overset{[u,v]}{\Longrightarrow} C$, we also add a lower-case edge $A \xrightarrow{c:u} C$ and an upper-case edge $C \xrightarrow{C:-v} A$.

In general, searching the labeled distance graph for an inconsistency means that we have to quantify over all possible values for the antecedent variables of the conditional statements. In other words, we have to quantify over all possible values of the contingent events. Under a brute force scheme, this yields an exponential number of possible combinations. Accordingly, the more that we eliminate conditional constraints, the less the algorithm has to branch and the easier it is to evaluate consistency.

Because the constraints and conditions of an STNU all involve events and because events are scheduled in temporal order, our algorithm can use this ordering to simplify our reasoning by taking conditional constraints and indicating how they apply unconditionally. Imagine we are considering a constraint of the form "if the contingent constraint ending at event F takes on its maximum possible duration, then event G must be scheduled by 10 minutes after event E at the latest." In addition, we separately know that we cannot infer the duration of the contingent constraint ending at F sooner than 25 minutes after E. In such a case, if we scheduled G more than 10 minutes after E, but before we learned the value of F, it may be possible that we learn that F takes on its maximum, causing us to violate our conditional constraint. In order to avoid this, we must *unconditionally* schedule G to occur no more than 10 minutes after E. This is because we have no way of learning whether the contingent constraint ending at F takes on its maximum duration in time for us to safely ignore the conditional constraint.

Furthermore, because each edge in a labeled distance graph corresponds to a constraint, we can use paths in the distance 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 more quickly explore the constraint space and determine whether it is possible to construct a schedule for our STNU. For any path through the graph, we can derive a new implicit inequality constraint based on the endpoints of that path with weight equal to the weight of the path and whose label is the union of all edge labels in the path.

To illustrate this process, we walk through Example 3.2 and demonstrate how to use the paths through individual constraints to determine that the entire system is uncontrollable (see Figure 4-2). In subsequent sections, we demonstrate how this procedure is automated using graph algorithms.

The simplest way to derive new constraints is in the same manner that we do for STNs. We take a set of edges along the path and produce a new edge whose start, end, and weight are the start, end, and accumulated weight of the path. In our

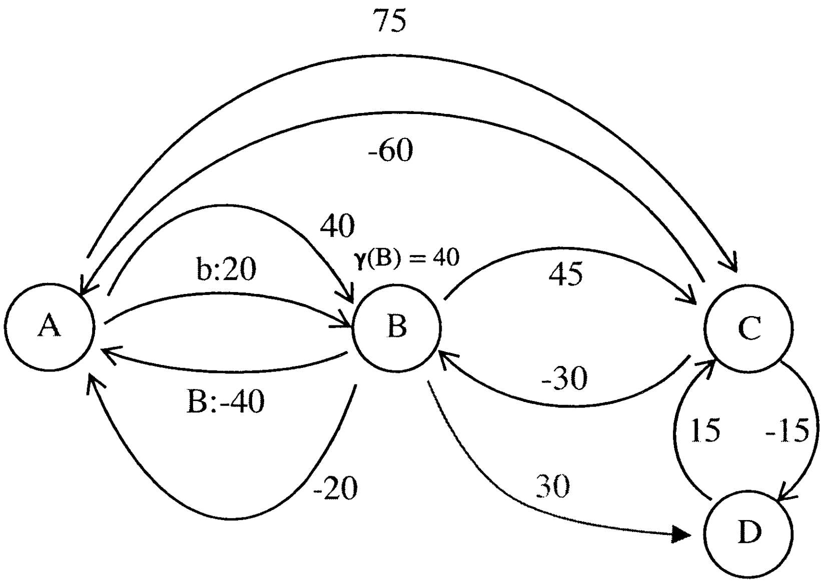


Figure 4-3: A recreation of Example 3.2 in labeled distance graph form after adding one new derived constraint.

example, we start by considering the path from B to D (see Figure 4-3). The path has two edges, $B \xrightarrow{45} C$ and $C \xrightarrow{-15} D$, and we construct $B \xrightarrow{30} D$. This denotes the inequality $D - B \leq 30$ and requires D to happen no more than 30 minutes after B, which is consistent with two constraints in the original STNU, "C must happen 30 to 45 minutes after B" and "D must happen exactly 15 minutes before C".

We also derive a new constraint between A and D. While it is possible to combine $A \xrightarrow{40} B$ with our newly derived edge $B \xrightarrow{30} D$, we get a tighter constraint if we use the lower-case edge, $A \xrightarrow{b:20} B$.

From $B \xrightarrow{-20} A$, we know that the earliest that B may happen is 20 minutes after A. In addition, from our derived constraint, D must happen no more than 30 minutes after B. Unfortunately, it is not possible for us to know exactly when B happens until 40 minutes after the fact, given that $\gamma(B) = 40$. As a result, we know that, if we schedule D to occur after we observe B, we will violate $B \xrightarrow{30} D$. Similarly, if we schedule D to occur more than 50 minutes after A, but assign D its value before we observe B, there is a chance that B actually happens at 20 minutes and we again violate $B \xrightarrow{30} D$. To achieve consistency, we must enforce that D is scheduled for no

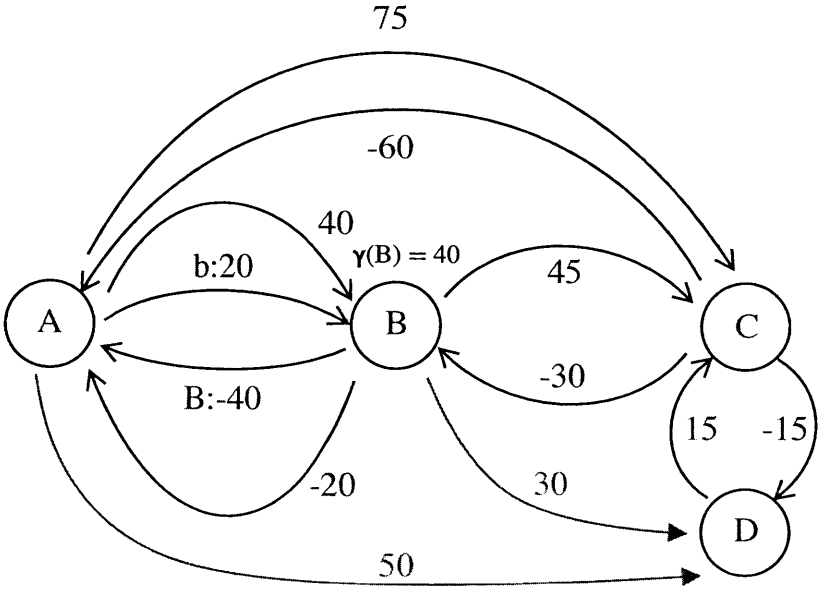


Figure 4-4: A recreation of Example 3.2 in labeled distance graph form after adding two new derived constraints.

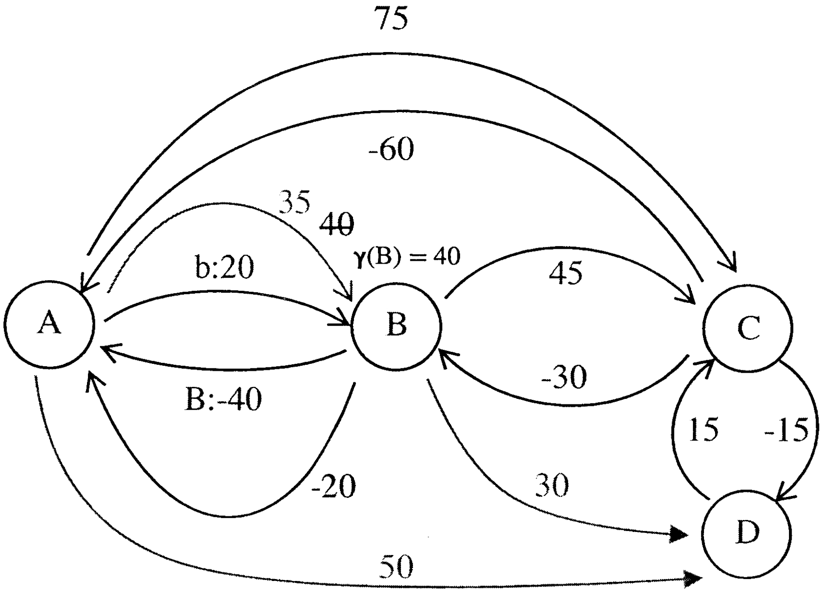


Figure 4-5: A recreation of Example 3.2 in labeled distance graph form after adding three new derived constraints.

more than 50 minutes after A. This constraint is represented as $A \xrightarrow{50} D$ (see Figure 4-4). In the next subsection we show how to derive these constraints automatically.

Finally, we show why this example is uncontrollable. We construct a new shortest path from A to B, using $A \xrightarrow{50} D$, that continues through $D \xrightarrow{15} C$ and $C \xrightarrow{-30} B$. This yields the edge $A \xrightarrow{35} B$, which tightens the old edge from A to B (see Figure 4-5).

This new constraint tells us that B must happen no more than 35 minutes after A. But B is a contingent event, hence not chosen by the scheduler; it depends on the actual length of the commute. The contingent constraint has an upper-bound of 40 minutes, which disagrees with the 35 minute upper-bound. We have a contradiction, and the network is uncontrollable.

We derive this contradiction by combining $A \xrightarrow{35} B$ and $B \xrightarrow{B:-40} A$, which yields a negative cycle in the graph that contains label B. This negative cycle corresponds to the conditional statement that whenever the contingent constraint ending at B takes on its maximum possible value, we have a contradiction. A system is controllable only if for every possible realization of contingent link durations, or for every ***projection***, it is possible to construct a valid schedule. As a result, given that we have found a possible projection in which we cannot construct a valid schedule, namely the one where $A \Rightarrow B$ takes on its maximum possible value, we know that the network is not delay controllable.

It is worth noting that finding this type of contradiction is equivalent to determining delay controllability and reduces to checking for the existence of a *semi-reducible negative cycle* on an STNU's labeled distance graph. This can be done in polynomial time. While we will shortly explain the details of semi-reducible negative cycles in much greater depth, our high-level aim is to draw an analogy between checking feasibility in STNs and STNUs.

It is worth noting that our approach for automatically recognizing inconsistencies will closely mirror the approach used to find inconsistencies in STNs. For STNs, we construct a distance graph and search for a negative cycle. For STNUs, we construct a labeled distance graph and search for a generalized version of a negative cycle, called a semi-reducible negative cycle. Before introducing semi-reducible negative cycles in detail, we show how to propagate our constraints and generate new edges,

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

Table 4.1: Edge generation rules for a labeled distance graph

representing valid constraints in a labeled distance graph.

4.2.1 Edge Generation Rules

In this section, we describe rules to derive new constraints from old ones, much like we did in the preceding section, but in a complete fashion. We use these generated constraints later to check for inconsistencies and evaluate the delay controllability of an STNU. For brevity, we use our edge notation as shorthand for the corresponding constraints.

To derive implied constraints and check for controllability for STNUs, Morris [40] introduced a set of edge generation rules, called reductions. Our key extension to these reductions, in order to handle delay controllability, is to include modifications that model non-zero observation delays of contingent events (see Table 4.1). In this section, we give an intuitive illustration of these rules and how they generate new valid constraints over the original STNU, while the formal proofs can be found in Appendix C. In section 4.3, we prove that this set of rules produces a complete set of derived constraints, in that they can always be used to determine the controllability of an STNU.

To illustrate the operation of these rules, we provide a series of example STNUs and indicate what new constraints we can generate from the existing set of constraints. Note that while these are valid illustrations of the above rules, these examples make many assumptions that are above and beyond what is required by the rules. These include the relative ordering of the events and whether particular constraints are requirement or contingent constraints.

We start with the *no-case* and *upper-case* rules. The no-case rule operates like the edge generation rule for STN distance graphs. By following a path in the distance graph of unlabeled edges, we construct a new constraint from the start, end, and accumulated weight of the path.

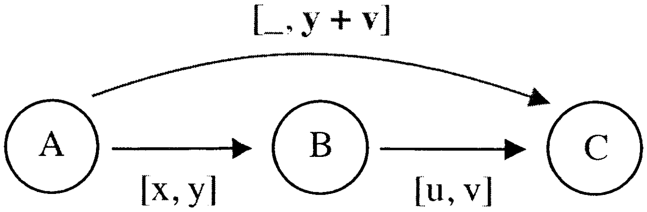


Figure 4-6: Example demonstrating the no-case rule.

Figure 4-6 illustrates a simple example involving the application of the no-case rule. By the figure, we know that B must happen at most y units of time after A and that C must happen at most v units of time after B. By the transitive property, this allows us to construct a new constraint from A to C, which says that C must happen no more than $y + v$ units of time after A. This is represented by the addition of the new constraint in blue, which has no lower-bound, but has upper-bound $y + v$.

Next, we consider the upper-case rule. The upper-case rule behaves like the no-case rule, but one of the edges has an upper-case label. The upper-case label enforces a condition on one of the constraints in the path, but because all other edges correspond to unconditional constraints, they also hold under the condition specified by the upper-case label, meaning that the entire path constraint holds under the upper-case label condition.

In Figure 4-7, we present an example with a single contingent link and a single requirement link to demonstrate the kind of inferences made by the upper-case rule. In this case, we use the upper-case rule to automatically make inferences about the lower-bound from A to C. We know that C must happen at least u units of time

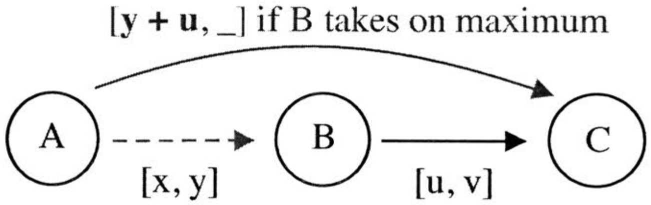


Figure 4-7: Example demonstrating the upper-case rule.

after B, but when B happens is outside of the scheduler's control. We know that C must always, unconditionally, happen at least $x + u$ units of time after A (we can derive this using the no-case rule). However, in certain instances, we do have a tighter lower-bound. Namely, in cases where the contingent link ending at B takes on its maximum possible duration, then C must happen at least $y + u$ units of time after A. This resulting conditional constraint, seen in blue in Figure 4-7, is automatically generated by the upper-case rule.

Next come the *lower-case* and *cross-case* rules. These are the first rules that strengthen conditional constraints to unconditional ones and the first ones that take into account the effects of delay.

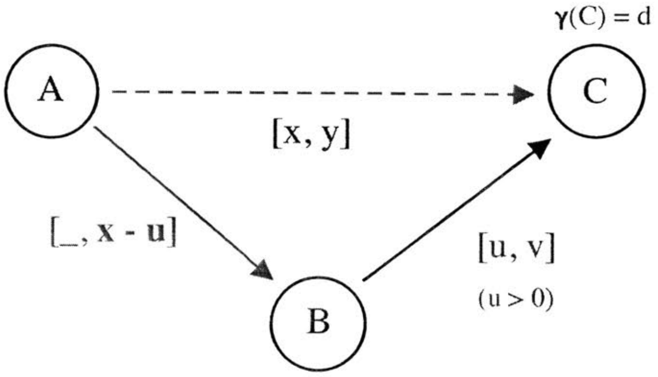


Figure 4-8: Example demonstrating the lower-case rule.

We start by providing an example application of the lower-case rule (see Figure 4-8). In this scenario, we have a contingent link from A to C and a requirement link constraining the relative timing of B and C. What makes this particular example notable is that B is guaranteed to happen before C is observed (this happens so long as $u > -d$, which is always the case if $u > 0$). The new constraint generated by the lower-case rule is that B must happen at most $x - u$ units of time after A.

In order to see why this constraint applies, it is useful to consider a proof by contradiction. Imagine that it were possible for B to occur more than $x - u$ time after A. In order for this to be permissible, all other constraints must also be respected, regardless of the duration of the contingent link ending at C. It is important to note that in this situation, the scheduler must always pick a time for event B before it sees when C happens. As a result, if it picks a time such that $B - A > x - u$, there is a possibility the scheduler eventually learns that C happened exactly x units of time after A. If we $C - A = x$ and subtract the equation $B - A > x - u$, we get $C - B < u$, which violates our original constraint. Thus, the constraint generated by the lower-case rule, $B - A \geq x - u$ holds unconditionally.

The cross-case rule follows the same logic as the lower-case rule. However, in this instance, we are combining two conditional edges, one upper-case and one lower-case, hence, the use of the term cross-case.

The final rule is the *label removal* rule. Like the lower-case and cross-case rules, the label removal rule eliminates a label by recognizing that we have to assign values to both events in a constraint before we can determine whether the antecedent represented by the label is true. Whereas in the previous two rules, we eliminated a lower-case label, with the label removal rule, we remove upper-case labels.

We present the label removal rule in Figure 4-9. We start by applying the upper-case rule to generate a conditional constraint between A and B (see Figure 4-9a). Note that the conditional constraint only applies if the contingent link ending at C takes on its maximum possible duration. The label removal rule, however, tells us that in this scenario it is safe to make the conditional constraint unconditional.

We illustrate as follows. If we know that the contingent link ending at C is not taking on its maximum possible duration, then we are free to ignore this constraint, as the antecedent is false. However, if we are scheduling A more than $v - y$ units of

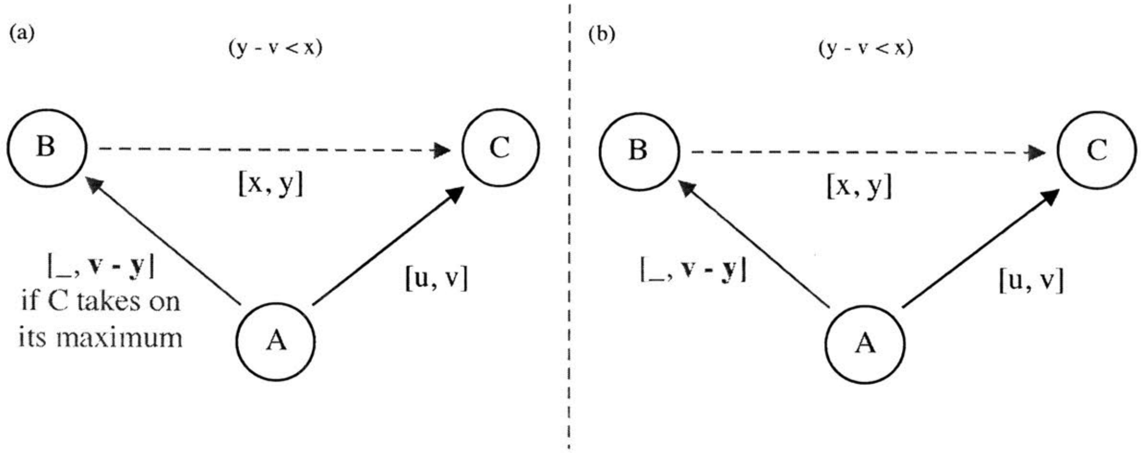


Figure 4-9: Example demonstrating the label removal rule. In part (a), we have the first new constraint we can add via a straightforward application of the upper-case rule. In part (b), given the assumption that $y - v < x$, we can remove the condition on the constraint as a result of the label removal rule.

time before B (and $v - y > -x$), then we will not yet have observed if the contingent link ending at C is taking on its maximum duration. Thus, in order to ensure that execution proceeds correctly, we must conservatively ensure that A is always scheduled no more than $v - y$ before B, to account for the possibility that the contingent link ending at C might take on its maximum possible value.

There are still additional rules that can be derived above and beyond the set of five provided so far. However, as we show in the following sections, this set of five rules is refutation complete; it is sufficient to apply these rules to quiescence to determine whether an STNU is delay controllable.

4.3 Verifying Delay Controllability

The purpose of applying these rules and generating additional constraints is to determine whether or not an STNU is controllable. After performing constraint propagation, we search for an inconsistency to tell us whether the STNU is in fact uncontrollable.

In the case of STNs, the presence of a negative cycle was sufficient to indicate that the temporal network is inconsistent. Unfortunately the presence of a negative

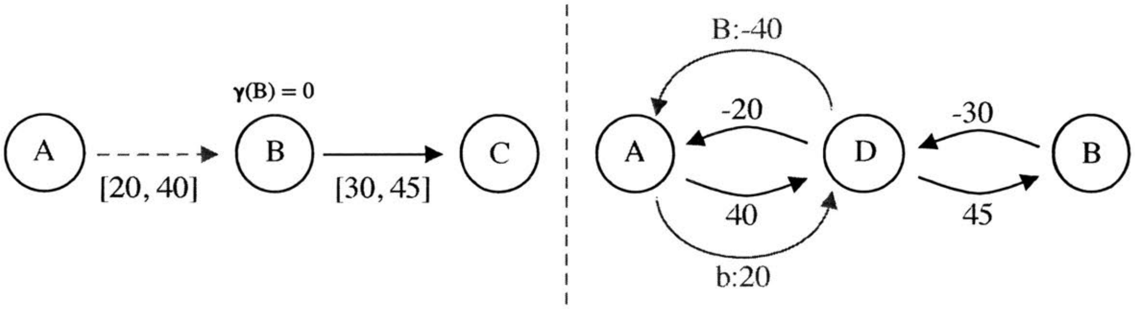


Figure 4-10: Example demonstrating the incorrect belief that the presence of a negative cycle implies that the network is uncontrollable. The STNU on the left is delay controllable, but the edges in blue in the labeled distance graph on the right form a negative cycle.

cycle through arbitrary sets of edges, both labeled and unlabeled, 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 negative cycle may exist that relies on two conditions that are never simultaneously true. Consider Figure 4-10; in this example, we have an STNU that is delay controllable, but its labeled distance graph has a negative cycle, as seen in blue. The reason this cycle does not indicate that the original STNU is uncontrollable is that one of the constraints of the negative cycle holds only if the contingent link $A \Rightarrow B$ takes on its minimum possible duration, whereas the other constraint only holds if the same contingent link takes on its maximum possible value.

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 [37].

Definition 4.1. Semi-Reducible Negative Cycle [37]
A labeled distance graph is said to have a semi-reducible negative cycle if the set of constraints represented by the edges of the labeled distance graph entails a set of edges that form a negative cycle with only unlabeled and upper-case edges.

When generalizing semi-reducible negative cycles for use in delay controllability, because the application of some of the edge generation rules depends on the specific delay function γ, we say that a negative cycle is semi-reducible with respect to γ.

If we can apply a series of transformations that ensures that the resulting negative cycle is free of lower-case edges, then we never encounter a situation where two different constraints fail to hold simultaneously. If we only have upper-case edges, then all constraints may hold if every contingent link takes on its maximum possible duration. Because controllability asks whether it is possible to construct a valid schedule in *all* possible projections, finding an inconsistency for one possible assignment of contingent link durations is sufficient to fail the delay controllability test.

4.4 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 determining dynamic controllability in STNUs [38].

4.4.1 Algorithm

Before explaining the operation of the algorithm in detail, we walk through this algorithm step-by-step with a simple example, before giving a sketch of its operation at a higher level.

```
Input: A labeled distance graph, $G = \langle V, E \rangle$; A delay function $\gamma$
Output: Whether the STNU derived from the distance graph is delay
        controllable with respect to $\gamma$
Initialization:
negNodes ← the set of all vertices with incoming negative edges;
DelayControllable?:
for $v \in$ negNodes do
    cycleFree? ←SRNCFree?($G$, $\gamma$, $v$, $[v]$, negNodes);
    if !cycleFree? then
        return false;
return true;
```

Algorithm 1: Delay Controllability algorithm

```
Input: Labeled distance graph $G = \langle V, E \rangle$, delay function $\gamma$, terminal node $s$,
       $callStack$, and negative nodes $negNodes$
Output: Whether the current walk is cycle-free
Initialization:
$Q \leftarrow PriorityQueue()$;
$distances \leftarrow []$;      # shortest distances for semi-reducible path;
$distances[\langle s, \emptyset \rangle] \leftarrow \langle 0, \emptyset \rangle$;
for $e \in s.incomingEdges()$ do
    if $e.weight < 0$ and $!e.lowerCase()$ then
        $Q.add(\langle e.from, e.label \rangle, e.weight)$;
        $distances[\langle e.from, e.label \rangle] \leftarrow \langle e.weight, e \rangle$
SRNCFree?:
if $s \in callStack[1 : end]$ then
    return $false$;
while $Q.size() > 0$ do
    $v, label, weight \leftarrow Q.pop()$;
    if $weight \geq 0$ then
        $G.add(\langle v, s, weight \rangle)$;
    else
        if $v \in negNodes$ then
            $newStack \leftarrow [v].concat(callStack)$;
            $result \leftarrow$ SRNCFREE?($G$, $\gamma$, $v$, $newStack$, $negNodes$);
            if $!result$ then
                return $false$;
        for $e \in v.incomingEdges()$ do
            if $e.weight \geq 0$ and ($!e.isLowerCase()$ or $e.label \neq label$) then
                $w \leftarrow e.weight + weight$;
                if $Q.addOrDecKey(\langle e.from, label \rangle, w)$ then
                    $distances[\langle e.from, label \rangle] \leftarrow \langle w, e \rangle$;
                $lower \leftarrow (e.from).incomingLowerEdge()$;
                if $lower \neq null$ and $e.weight < \gamma(lower.label)$ then
                    if $Q.addOrDecKey(\langle lower.from, label \rangle, w + lower.weight)$
                     then
                        $distances[\langle lower.from, label \rangle] \leftarrow$
                          $\langle w + lower.weight, lower \rangle$;
$negNodes.remove(s)$;
return $true$;
```

Algorithm 2: Function SRNCFREE?

Example Walkthrough

To illustrate how the algorithm works, we perform delay controllability checking on the example in Figure 4-11, which contains a subset of the constraints from Example

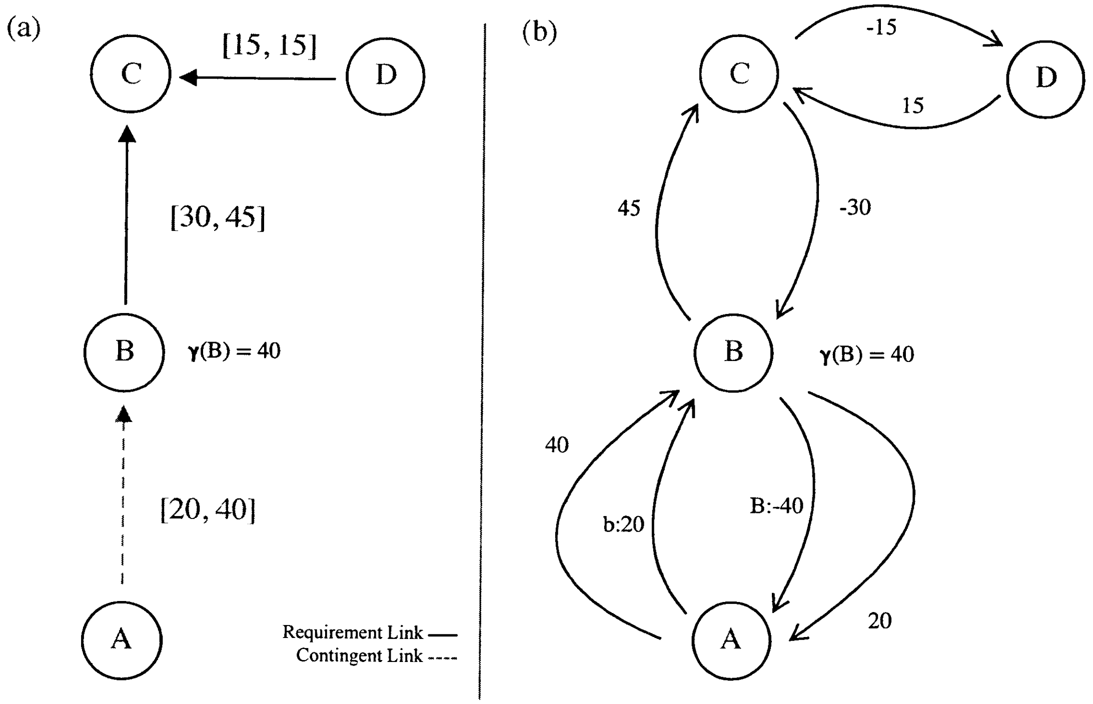


Figure 4-11: (a) Example of an STNU that is uncontrollable when $\gamma(B) = 40$. (b) The same STNU represented as a labeled distance graph.

3.2 that together still render the network uncontrollable.

The algorithm start by collecting all nodes that have incoming negative edges in the labeled distance graph (Algorithm 1, line 1), which gives us $[A, B, D]$, and starts performing SRNCFREE? on the nodes of the list (lines 2 & 3; see Figure 4-12). For this example, we assume that the algorithm arbitrarily picks node A to start and enqueues all incoming edges with negative weight (Algorithm 2, lines 4-7), before starting its walk through the labeled distance graph, using a variant of Dijkstra's algorithm (lines 10-28).

It is important to note that the algorithm walks through the graph backwards (see line 20 for the reference to incoming edges). It does so in order to simplify the act of picking successor edges, which are used to guarantee that walks are only along semi-reducible paths. Specifically, a lower-case edge with label c can only be reduced if followed by an edge with weight less than $\gamma(C)$. It is difficult to look-ahead and see whether some set of reductions could produce an edge with low enough weight to

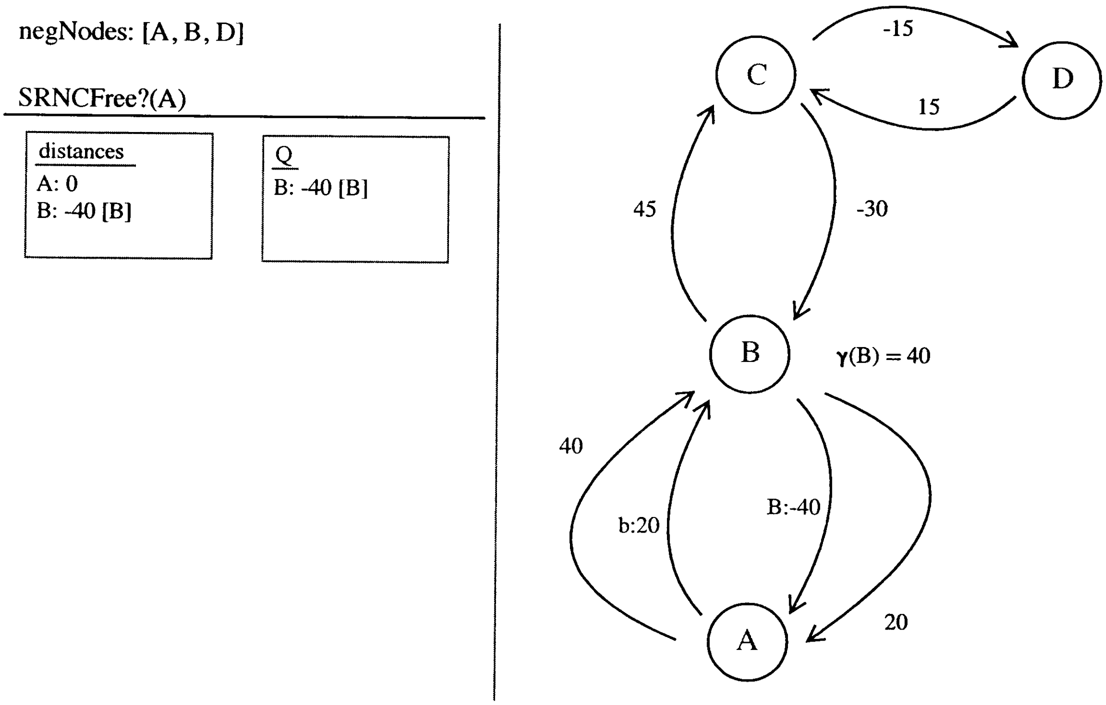


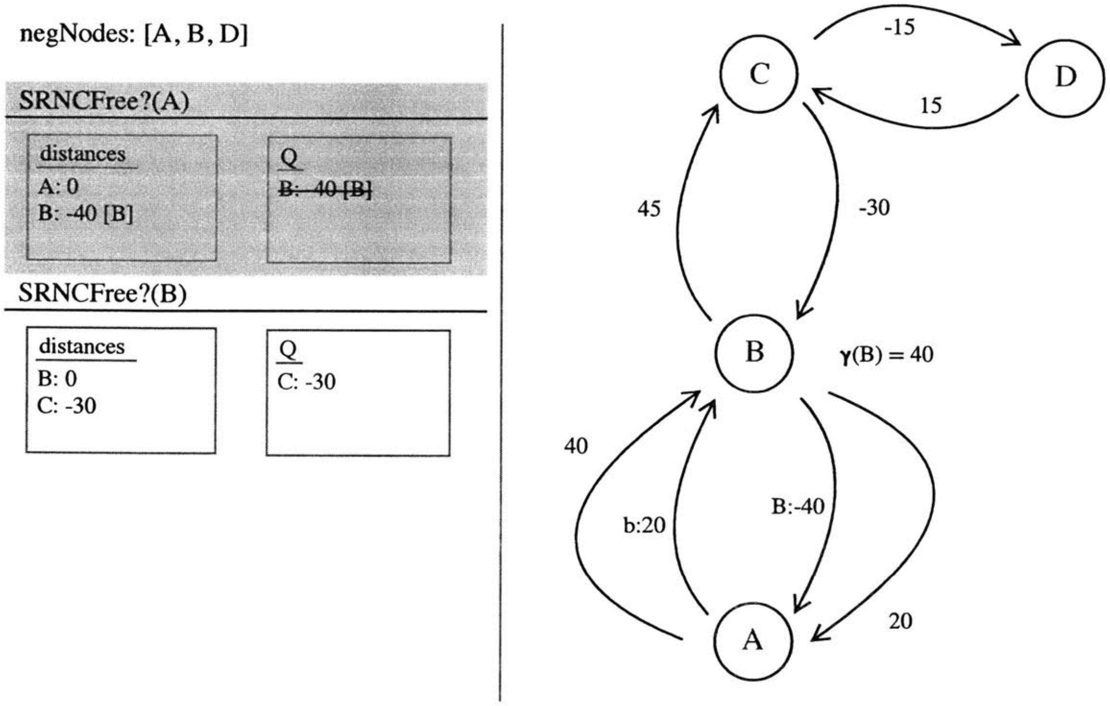
Figure 4-12: Step 1 of the walkthrough. SRNCFree? called with node A.

apply the lower-case or cross-case rules, but when walking in reverse, it is easy to see that the weight of the path that has been walked so far comes under the lower-case rule's threshold $\gamma(C)$.

The first node dequeued by the algorithm is B (line 11), which is reached using $B \xrightarrow{B:-40} A$, and because B has incoming negative edges, we recurse (lines 15, 17; see Figure 4-13).

The algorithm repeats this process again by calling SRNCFree? with B and dequeues the node C (line 11) before adding its successors (lines 20-28). Next to be dequeued is D (line 11), which is of distance -15 away from B (see Figure 4-14. Because D is also a negative node, the algorithm again recurses (line 17; see Figure 4-15).

With the recursive call to D, SRNCFree? terminates for the first time. C is enqueued (lines 4-7) using $C \xrightarrow{-15} D$, and when C is later dequeued (line 11), the only edge that provides a new, shorter path is $B \xrightarrow{45} C$ (see Figure 4-16). When B is



negNodes: [A, B, D]

SRNCFree?(A)

distances
A: 0
B: -40 [B]

Q
~~B: -40 [B]~~

SRNCFree?(B)

distances
B: 0
C: -30

Q
C: -30

Figure 4-13: Step 2 of the walkthrough. We recurse and call SRNCFREE? with node B.

reached, SRNCFREE? terminates and adds $B \xrightarrow{30} D$ to the graph as a newly derived semi-reducible path (lines 12-13; see Figure 4-17). The recursive call then returns and resumes the stack frame that had terminal node B (line 30; see Figure 4-18).

In the previous recursive call, D had just been reached, so now the algorithm is about to consider the additional edges it can take to continue the path. One of the edges that could be considered is the newly added $B \xrightarrow{30} D$. However, this extension would not be enqueued, 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. However, the algorithm still must perform the lookahead at lines 25-28. Since the weight of the new $B \xrightarrow{30} D$ is less than $\gamma(B)$, the lower-case reduction is implicitly applied to create $A \xrightarrow{50} D$ from $A \xrightarrow{b:20} B$ and $B \xrightarrow{30} D$ (see Figure 4-19). When the continuation of Dijkstra's algorithm eventually reaches A, the edge $A \xrightarrow{35} B$ is newly added (line 13; see Figure 4-20) and return (line 30).

The algorithm finally returns to our original recursive call that was invoked with

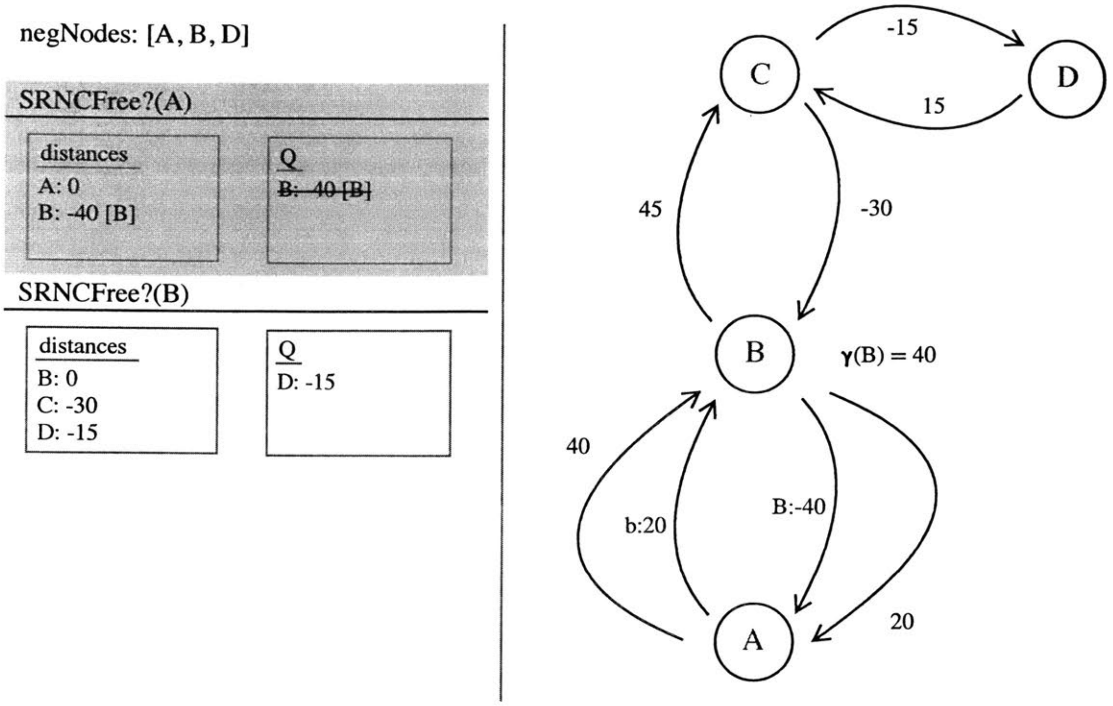


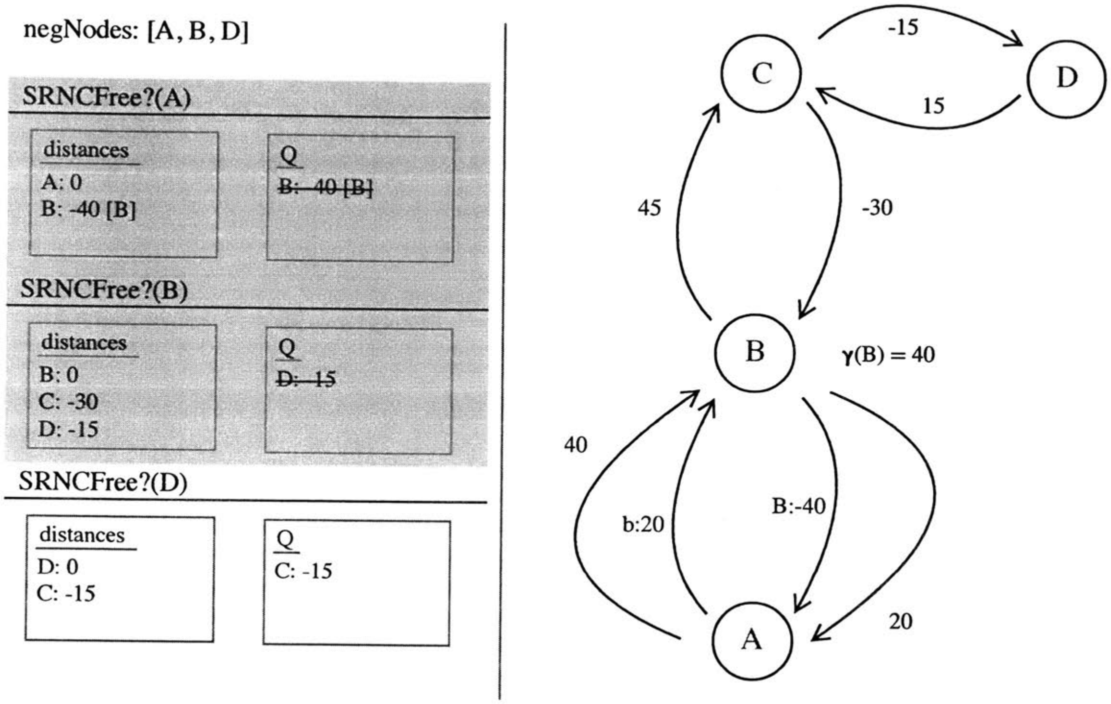
Figure 4-14: Step 3 of the walkthrough. We dequeue $C \xrightarrow{-30} B$ and enqueue $D \xrightarrow{15} C$.

node A (see Figure 4-21). The algorithm was about to consider B's extensions (lines 20-28) and now has the new edge $A \xrightarrow{35} B$. But by taking that edge, the algorithm returns back to A, since the path of $B \xrightarrow{B:-40} A$ and $A \xrightarrow{35} B$ has total weight -5. A has incoming negative edges; thus, the algorithm recurses again. However, because A was already in the call stack (line 8), the algorithm returns false, meaning the STNU is recognized as not delay controllable (see Figure 4-22).

Algorithm Sketch

We now give a high-level sketch of how the algorithm operates and why it is designed in the way that it is. It is most useful to start by considering the operation of SRNCFREE? (Algorithm 2).

SRNCFREE? implements a variant of Dijkstra's algorithm, performing a single-destination shortest path search to find the shortest semi-reducible path to each initial terminal node s in the graph. Note that whenever a semi-reducible negative


negNodes: [A, B, D]

SRNCFree?(A)

distances
A: 0
B: -40 [B]

Q
~~B: -40 [B]~~

SRNCFree?(B)

distances
B: 0
C: -30
D: -15

Q
~~D: -15~~

SRNCFree?(D)

distances
D: 0
C: -15

Q
C: -15

Figure 4-15: Step 4 of the walkthrough. We recurse and call SRNCFREE? with node D.

cycle exists, the concept of a shortest semi-reducible path between two nodes is not well-defined. Hence, if our search for shortest semi-reducible paths starting from some node fails, we know that there must be a semi-reducible negative cycle in the labeled distance graph. Accordingly, Algorithm 1 serves mainly as a wrapper around SRNCFREE? in order to guarantee that SRNCFREE? is called from every relevant node in the graph to check that the shortest semi-reducible paths are well-defined everywhere.

It is worth noting that Dijkstra's algorithm is incapable of handling negative edges. To accommodate this restriction, SRNCFREE? only considers those negative edges connected to the original input node s and discards all others; if there exists a guarantee that the only possible negative edge in a path is the very first one, then Dijkstra's algorithm operates just fine.

The algorithm works by walking along paths from the initial node s, continuing its walk using non-negative edges until there are no edges it can take along its walk

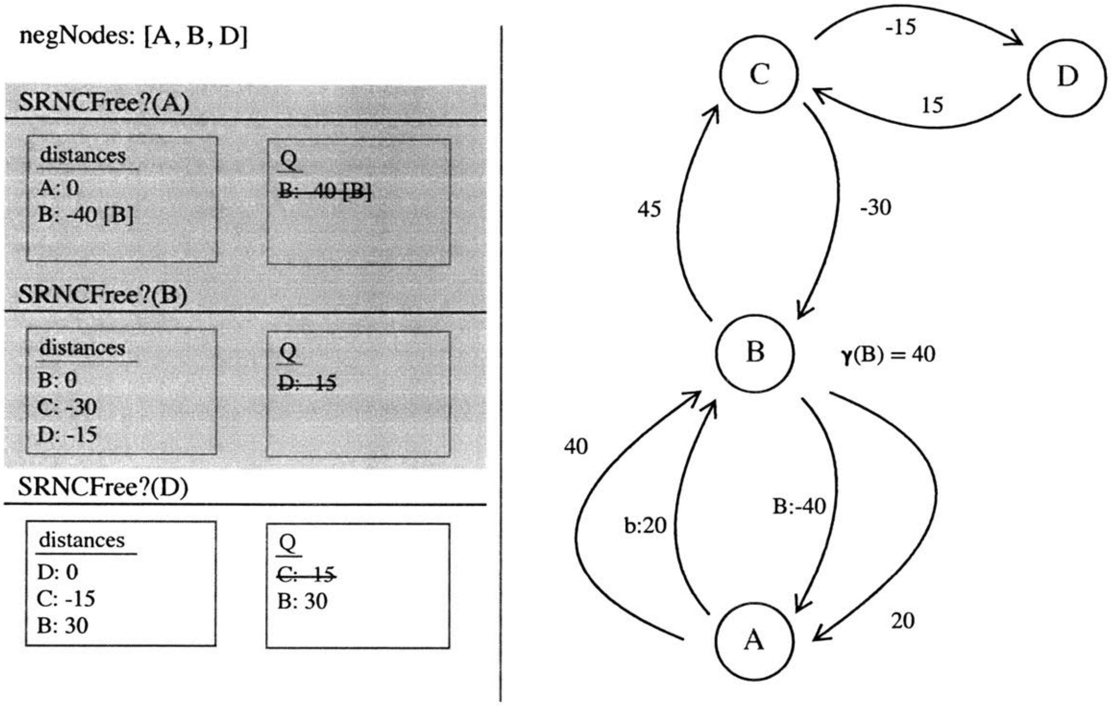


Figure 4-16: Step 5 of the walkthrough. We dequeue $C \xrightarrow{-15} D$ and enqueue $C \xrightarrow{45} B$.

or the weight of the path eventually become positive. The algorithm walks edges in reverse from terminal node s to all other nodes, in order to simplify the act of picking successor edges, to guarantee that walks are only along semi-reducible paths. Specifically, a lower-case edge with label c can only be reduced if followed by an edge with weight less than $\gamma(C)$. 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 the weight of the path that has been walked so far comes under the lower-case rule's threshold $\gamma(C)$. If 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 in that it does not consider negative edges except those connected to initial node s. Whenever the algorithm reaches a node that has an incoming edge with negative weight, it recursively calls SRNCFREE? on that node to begin a new sub-walk whose only negative edge is con-

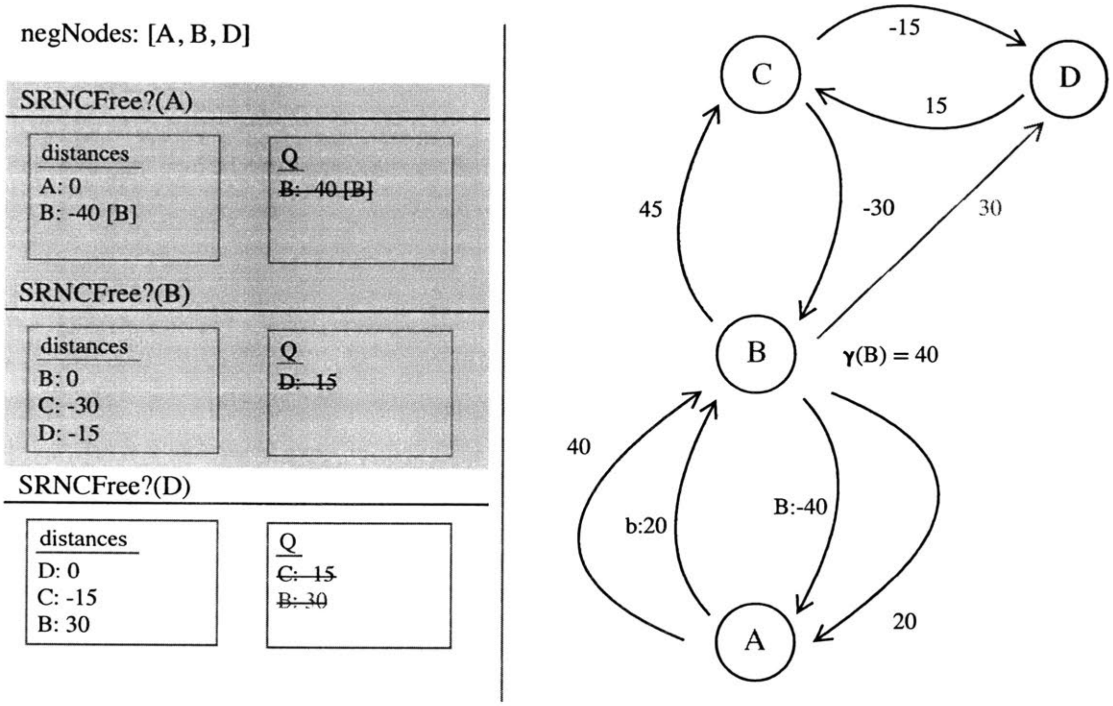


Figure 4-17: Step 6 of the walkthrough. Edge $B \xrightarrow{30} D$ (in blue) is added to the labeled distance graph.

nected to the sub-walk's initial node. The recursive call either eventually completes successfully (potentially involving subsequent recursive calls), in which case all of the non-negative extensions of that edge have been added to the graph, or it continues recursing indefinitely as its walk continues. If we were to infinitely recurse, we know we have found a semi-reducible negative cycle, since each recursive sub-call found a semi-reducible negative path, and all those semi-reducible negative paths when combined form a semi-reducible negative cycle. Note that this algorithm never actually enters infinite recursion, as it maintains the set of nodes s that have been considered recursively in the variable *callStack* and checks whether the same terminal node s has already been considered (line 8 of Algorithm 2).

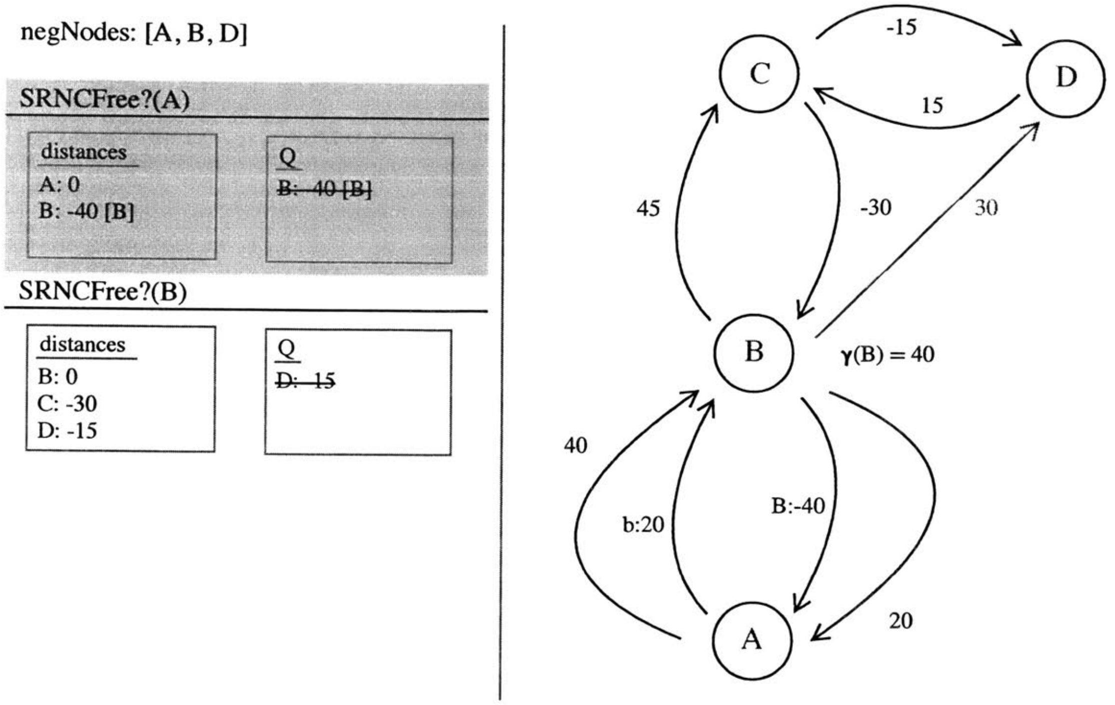


Figure 4-18: Step 7 of the walkthrough. We return to the previous recursive call that was invoked with node B.

4.4.2 Correctness

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

Theorem 4.1. *Algorithm 1 is sound and complete with respect to determining whether an STNU's labeled distance graph is free of semi-reducible negative cycles with respect to delay function γ.*

Proof. By Lemmas 4.2 and 4.3, we know that Algorithm 1 is sound and complete. □

We prove this in parts by proving each direction of the if and only if independently. We start by considering what happens when the algorithm returns false.

Lemma 4.2. *If Algorithm 1 returns false on STNU S and delay function γ, then S has a semi-reducible negative cycle with respect to γ.*

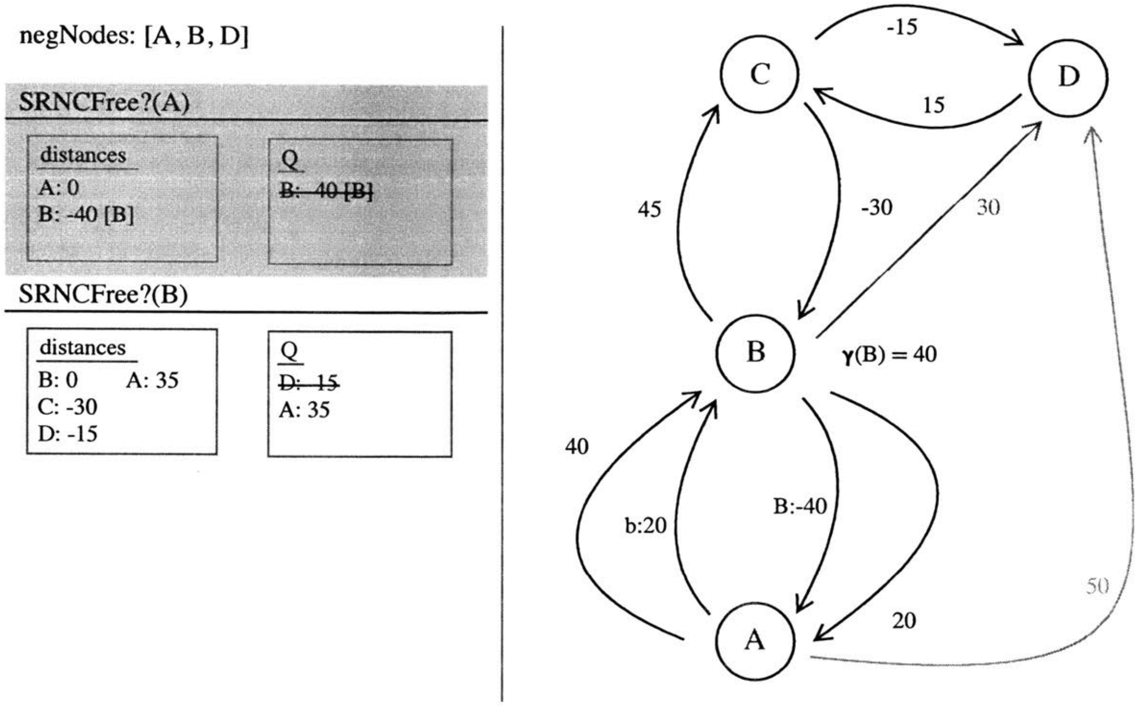


Figure 4-19: Step 8 of the walkthrough. New edge $A \xrightarrow{50} D$ (in green) is derived by performing a lower-case reduction combining $A \xrightarrow{b:20} B$ and $B \xrightarrow{30} D$. The new edge is only used implicitly in this stack frame and is not added to the labeled distance graph.

Proof. Algorithm 1 only returns false if a recursive call to SRNCFree? returns false, which only happens if terminal node s was already present in the call stack. 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 SRNCFree?. 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

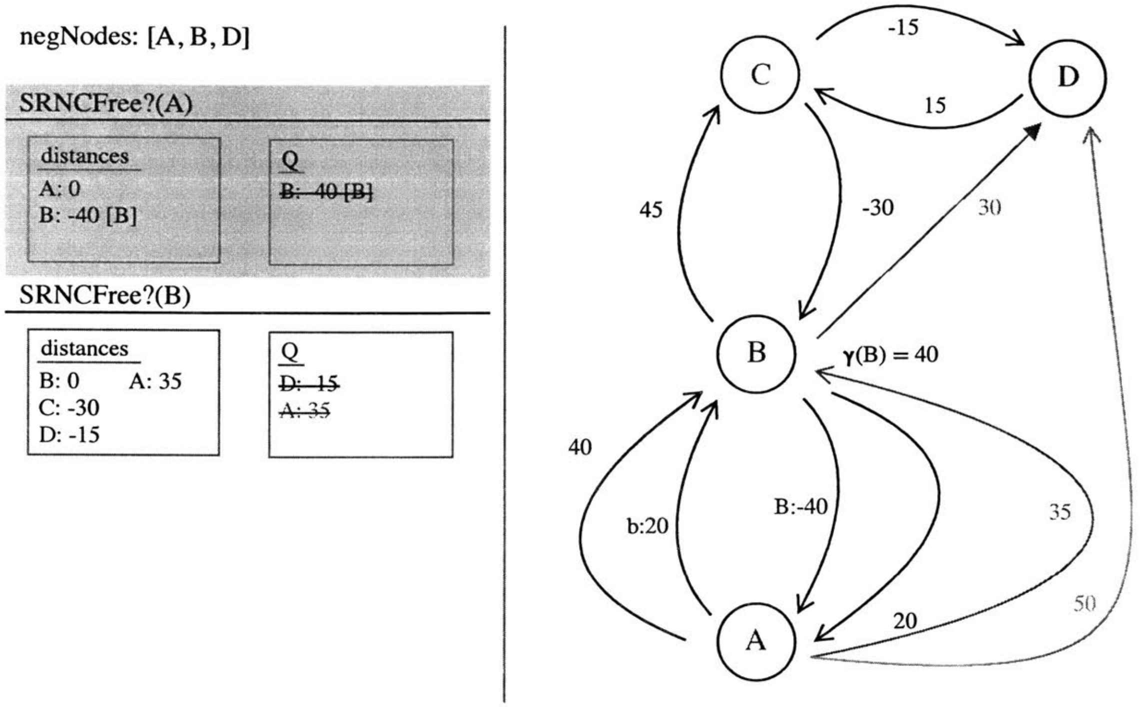


Figure 4-20: Step 9 of the walkthrough. Edge $A \xrightarrow{35} B$ (in blue) is added to the labeled distance graph.

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. Because all upper-case edges have negative weight, we know our path never adopts 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 to 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 SRNCFREE? 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 perform 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

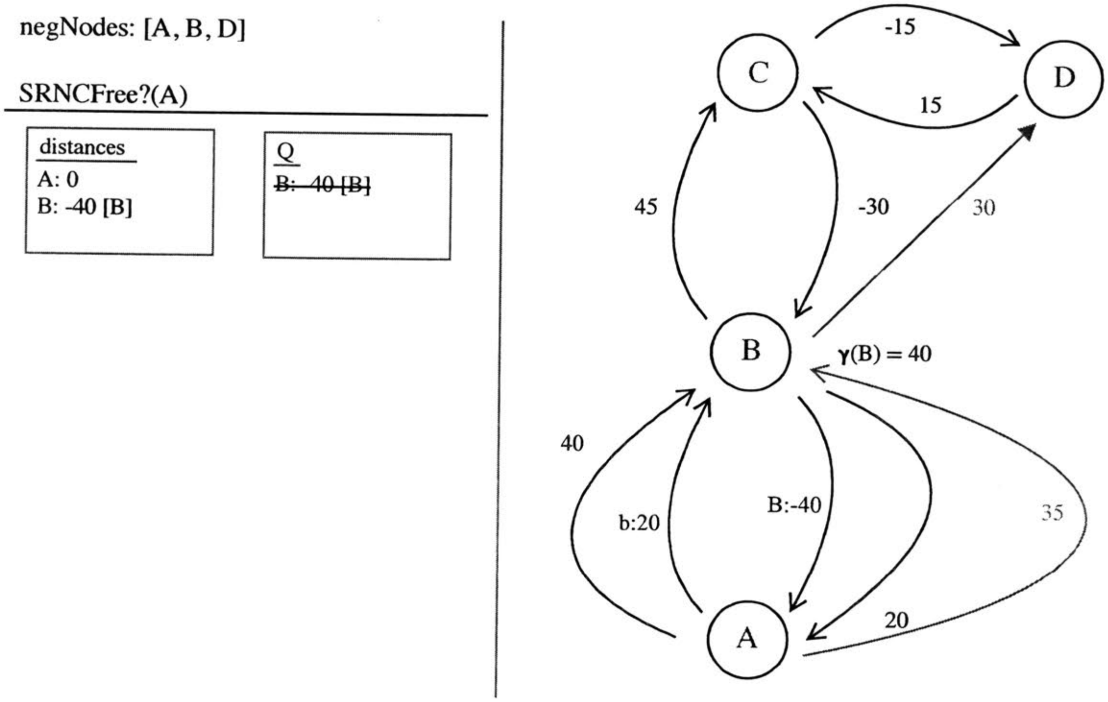


Figure 4-21: Step 10 of the walkthrough. We return to the previous recursive call that was invoked with node A.

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 our proof, we must also show that if the labeled distance graph implied by an STNU and delay function γ has a semi-reducible negative cycle, then our algorithm will find one.

Lemma 4.3. *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.

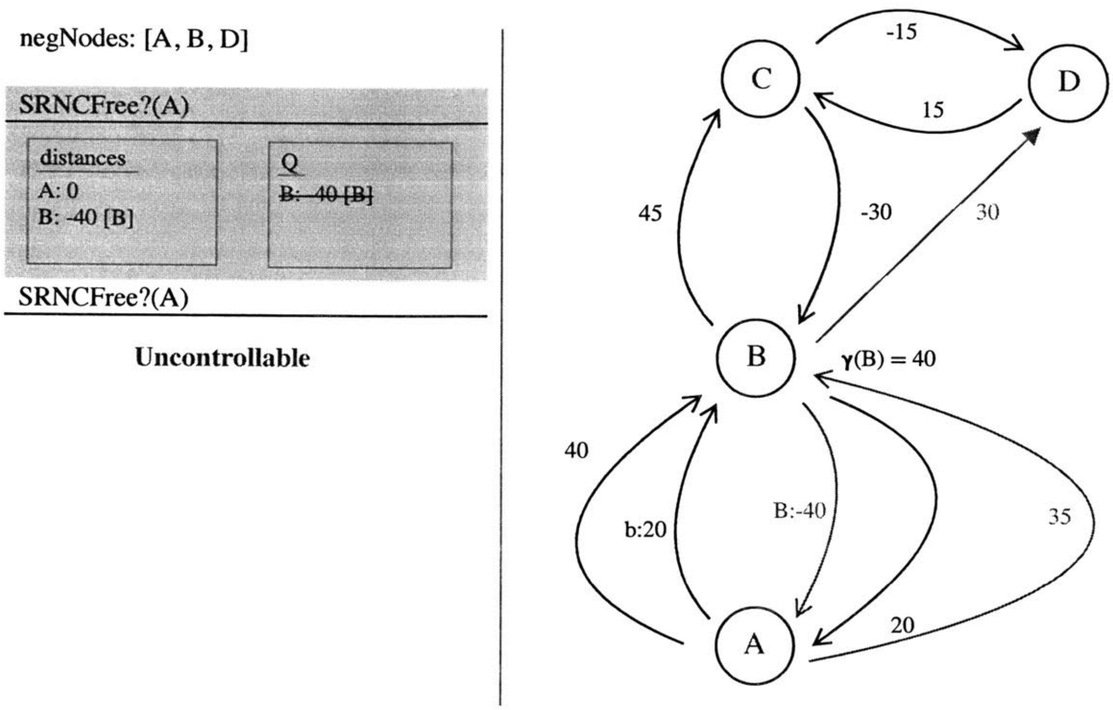


Figure 4-22: Step 11 of the walkthrough. We invoke SRNCFREE? again with node A, which implies the STNU is not delay controllable with respect to γ. The edges in red represent the semi-reducible negative cycle that can be extracted.

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 such that when we walk backwards from that edge along the semi-reducible path, the path weight becomes non-negative. We know that if SRNCFREE? 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 find the path with shortest weight. In our semi-reducible negative cycle, we 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 want 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 SRNCFree? from any one of them, our algorithm would return false, since the successful completion of any one SRNCFree? 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 eventually call SRNCFree? for all nodes with negative incoming edges, we have a guarantee that if there is a semi-reducible negative cycle, we return false. □

Stringing the two lemmas together, Theorem 4.1 concludes that our algorithm is sound and complete. What remains is to show is that the algorithm completes in $O(n^3)$ time.

Theorem 4.4. *Algorithm 1 operates in $O(n^3)$ time.*

Proof. First note that we only call SRNCFREE? 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 SRNCFREE? returns true for terminal node s, we know it has been removed from *negNodes* (line 29 of Algorithm 2), and whenever SRNCFREE? 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 SRNCFREE? at most n times in total.

Now we analyze the runtime of SRNCFREE? 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 SRNCFREE? 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 together, we run SRNCFREE? at most n times, giving us a total worst-case runtime of $O(n^3)$. □

4.5 Semi-Reducible Negative Cycles and Controllability

In this section, we provide a proof indicating that an STNU is delay controllable with respect to delay function γ if and only if its labeled distance graph has no semi-reducible negative cycles with respect to γ. We start by showing that the presence of a semi-reducible negative cycle proves that the corresponding STNU is not delay controllable. To demonstrate the other direction, we show that the absence of a semi-reducible negative cycle means that it is possible to construct a valid execution strategy for all projections under the given observation delay; if we can construct a valid execution strategy, then by definition, the STNU is delay controllable. Our main theorem is as follows:

Theorem 4.5. *Delay controllability of an STNU S with respect to γ can be checked in $O(n^3)$ time.*

Proof. With Lemma 4.6 and Lemma 4.9, we show that S is delay controllable if and only if S is free of semi-reducible negative cycles. By Theorem 4.1, we show 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. □

4.5.1 Semi-Reducible Negative Cycles Imply Uncontrollability

Lemma 4.6. *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. Given a semi-reducible negative cycle, we know that the cycle can be transformed into a cycle without lower-case edges by some combination of reductions. Consider the semi-reducible negative cycle after those reductions.

Given the new cycle, we can continue to combine all of its unlabeled edges together using the no-case reduction or combine unlabeled edges with upper-case edges using the upper-case reduction. Since each reduction reduces 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.

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 constraint was of the form $A \overset{[x,y]}{\Longrightarrow} C$. This means that, if we have not yet observed that C has occurred (which we will have not before execution), we may be required to enforce the constraint $A - B \leq u$. 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 constraints have yet occurred, so there exists a possible projection in which all upper-case constraints are active. In particular, it is the instance in which all contingent links take on their maximum possible duration, where all constraints in the cycle will hold. If we sum the left and right hand sides of the constraints, the left side will telescope to zero, and the right will reduce to the weight of the cycle, which is negative. If we take w to be the weight of the cycle, this gives us $0 \leq w < 0$, yielding a contradiction.

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

4.5.2 Finding an Execution Strategy

In order to show the converse, that a STNU whose labeled distance graph is free of semi-reducible negative cycles is delay controllable with respect to γ, we show that in those instances we can always construct an execution strategy that satisfies all constraints after observing the outcomes of contingent constraints only after their specified delay has elapsed. Our process for generating an execution strategy closely follows that of dynamic controllability by Hunsberger [28], as well as its proof of correctness.

Our execution strategy is based on computing the shortest semi-reducible paths from a simulated start event Z to all other events. If we find such paths for all events, then our strategy is to schedule the next available event, like we do for STNs, according to the bounds given by the shortest semi-reducible paths.

Before we elaborate our execution strategy in greater depth, we need to prove a few properties of labeled distance graphs that have no semi-reducible negative cycles. We do this to show that the notion of a shortest semi-reducible path in these labeled distance graphs is well-defined.

First, we show that the number of possible edges we can generate with our edge generation rules is bounded.

Lemma 4.7. *If an STNU's labeled distance graph does not have a semi-reducible negative cycle with respect to delay function γ, then a finite number of edges can be generated from the original STNU's labeled distance graph, with the edge generation rules from Table 4.1. 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 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 is 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, 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, which 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 output those two edges round i and round 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 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 forward or backward, 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 each edge of $B \rightsquigarrow B$ acts as the successor edge in one of the four binary edge generation rules since each edge is eventually combined with e_i. We notice that the only valid edges that can act as successor edges in the reduction rules 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 $\forall x_c \in X_c, \gamma(x_c) \geq 0$, 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 sub-cycle is semi-reducible.

If edge generation is unbounded, we have a series of transformations that transform e_i into e_j and can find a semi-reducible negative cycle. Thus, if there are no semi-reducible negative cycles, then the edge generation rules only produce a finite number of edges, all of which can be found after $O(n^3)$ rounds.

□

Knowing that there is a limit to the number of possible edges that can be generated is useful for determining whether the notion of a *shortest semi-reducible path* with respect to delay function γ is well-defined for an STNU's labeled distance graph. In other words, for each pair of nodes A and B, there is some value k, such that no semi-reducible path from A to B is shorter than k in the STNU's labeled distance graph. We prove that shortest semi-reducible paths are well-defined whenever a labeled distance graph is free of semi-reducible negative cycles.

Lemma 4.8. *If an STNU's labeled distance graph does not have a semi-reducible negative cycle with respect to some delay function γ, then for every pair of nodes A, B in the labeled distance graph, there exists some k that is less than the length of the shortest semi-reducible path between A and B. Hence, the notion of a shortest semi-reducible path is well defined.*

Proof. Assume that our labeled distance graph does not have a semi-reducible negative cycle with respect to γ. By Lemma 4.7, we know that there is a finite number of edges that can possibly be generated by the edge generation rules.

Examine any semi-reducible path between A and B. Since the path is semi-reducible, we know that we can apply a set of edge generation rules to yield a path with the same weight that does not have 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 bound in weight, since there are only finitely many possible edges that can be generated by the rules and thus that can be picked for use in the path. Thus, if an STNU is free of semi-reducible negative cycles, then there is a shortest semi-reducible path between all nodes.

□

Whenever an STNU has a valid execution 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. Because shortest semi-reducible paths are computable and well-defined, we can use them in an execution strategy for STNUs to define bounds on when individual events should be executed. Much of this proof resembles the one from [28], but special care is taken to show that the introduction of delay does not introduce new problems in the main arguments of the proof.

Lemma 4.9. *If the labeled distance graph of an 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 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 executable event's upper-bound for possible execution.

We derive the lower- and upper-bounds for an event's execution by looking at the shortest semi-reducible path between Z and other nodes, where Z represents the earliest occurring executable event, 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 4.8, 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 event to execute as follows. First, we search across all executable events that have yet to be assigned and find the ones with the earliest lower-bounds. If the selected events have lower-bound greater than t, our plan is to execute them at their lower-bound. Otherwise, that event is executed immediately (at time t), as our model allows for instantaneous execution. Every time an event is executed or a contingent event is observed, we re-determine what the next event to execute is. After all executable events have been specified, we reveal the remaining unobserved contingent events 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 constraint $A \overset{[x,y]}{\Longrightarrow} C$, 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 constraint 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 contingent event. Then we consider what happens when we execute an executable event. Whenever we know the new value of an event B, whether we executed or observed it, that occurred at time t, we fix that point in our labeled distance graph by adding constraints $Z \xrightarrow{t} B$ and $B \xrightarrow{-t} Z$. We show that in either case, if the original graph was free of semi-reducible negative cycles, then the resulting graph is also free of semi-reducible negative cycles. We also show that at every step along the way, no unexecuted event's upper-bound ever drops below the current time.

Case 1 - Contingent Event. 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 executable event A must have been scheduled before we observed the value of C. Say that A occurred at time τ. We can now add constraints between Z and C of weight t and $-t$. In addition to adding the two new constraints, we remove the $A \xrightarrow{c:x} C$ constraint and the $C \xrightarrow{C:-y} A$ constraint. Those constraints 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 constraint, 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 constraints create a semi-reducible negative cycle.

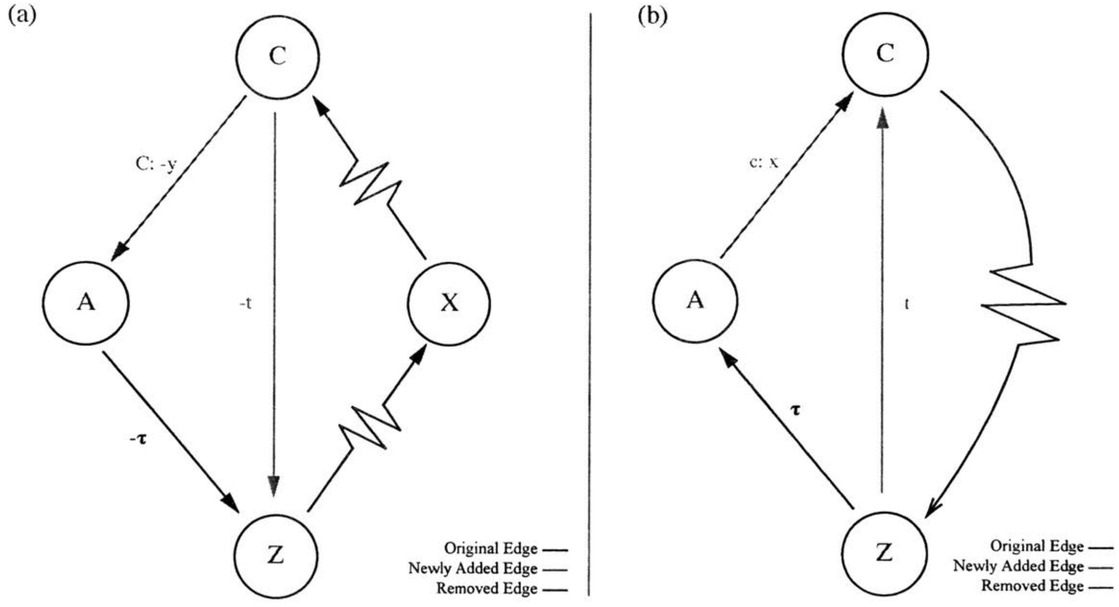


Figure 4-23: (a) Diagram indicating that the observation of C occurring at time t creates no semi-reducible negative cycles that use $C \xrightarrow{-t} Z$. The path $X \rightsquigarrow C$ starts with a lower-case edge if it is non-empty. (b) Another diagram indicating the same thing for the other added 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 4-23a).

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 $Z \rightsquigarrow C$. 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, as 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. We would then 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 4-23b). 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 $\beta > \gamma(C) \geq 0 \geq -D$, 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 drops 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 a semi-reducible negative cycle and so whenever we assign a contingent event, we do not create any new semi-reducible negative cycles.

We also know that no unexecuted event's upper-bound ever dips 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 does 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 that combines $A \xrightarrow{c:x} C$ 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 contingent event does not cause the upper-bound of an unexecuted event to drop below the current time.

Case 2 - Executable Event. Assume that we just assigned a value to executable event 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 event can have an upper-bound less than the current time, we have a contradiction. Thus, the assignment of an executable event 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 event 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 observe or execute an event, we keep our STNU free of semi-reducible negative cycles and preserve the executability of our STNU, since no executable event'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. □

Putting these lemmas together, we see that an STNU S is delay controllable with respect to γ if and only if S is free of semi-reducible negative cycles. In the next section, we conclude the proof of Theorem 4.5 by showing that we can check for the presence of semi-reducible negative cycles in $O(n^3)$ time.

4.6 Experimental Results

In this section, we present our experimental evaluation of delay controllability. We start by comparing strong, dynamic, and delay controllability against one another, focusing on the respective correctness of the algorithms. We show that dynamic controllability often incorrectly marks certain situations as controllable due to its failure to consider limitations in communication, and that strong controllability, while correct, is often overly conservative.

We then consider the practicality of using delay controllability by examining the runtime of our algorithm. We show that delay controllability runs fast enough to allow on-board evaluation in vehicles that have limited computational power. We also compare the runtimes of delay controllability to dynamic and strong controllability. The results show that delay controllability is an adequate replacement for dynamic controllability in virtually all settings and that mixed strategies (involving checking strong controllability first and then checking for delay controllability on failure) may be worth pursuing, depending on the expected distribution of input temporal networks.

4.6.1 Setup

Across our set of experiments, we evaluate the performance of delay controllability across two different sets of STNUs. The first is a set of randomly derived STNUs intended to be representative of a wide range of possible STNUs. The second is a set of STNUs meant to model AUV deployments. This second set is highly structured and scales naturally by altering the number of AUVs and the length of each AUV's mission.

In order to capture the difference in correctness across the different controllability checks, we evaluate the outcomes of these checks on a set of randomly generated STNUs. Our choice to use random STNUs was based on the desire to evaluate our results across a broad range of possible STNU structures that were representative of those that we might see in practice. Each random STNU has 10 disconnected contingent constraints 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 constraint endpoints, we created a requirement constraint between the two with probability $\frac{1}{40}$. Each requirement constraint has a lower-bound of 0 and an integer upper-bound uniformly chosen between 1 and 4. These parameters were experimentally tuned in order to ensure that there was a good mix of controllable and uncontrollable problems. We select these specific parameters because they represent a reasonable trade-off between simplicity in degenerate cases and sufficient complexity to exhibit interesting behaviors.

To evaluate our algorithm's runtime at scale, we modify the autonomous underwater vehicle (AUV) example used in previous works to evaluate dynamic controllability algorithms [6]. These were also used in the thesis work presented by [61]. In the AUV scenario, multiple vehicles navigate to a few different sites, parameterized by an uncontrollable duration, and spend some time collecting data at each location, an activity whose exact duration is specified by the AUV. To extend the example for delay controllability, we opted to simulate a series of planned communication outages throughout the course of AUV operation. We randomly assigned each contingent constraint 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 pre-compiled script, instead of waiting for instruction from a central operator.

For the purposes of our experiments, we uniformly randomly sampled the time taken to navigate 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 controllable. By adding in the correction term, in expectation it is infeasible to complete one of the scheduled activities if the corresponding navigation task is unobserved; this ensures that our set of temporal networks has a reasonable mix of controllable and uncontrollable networks. There is an additional global temporal constraint enforcing that each AUV must finish its activities within $5000 \cdot a$ units of time to simulate mission-wide deadlines.

4.6.2 Results

In the random STNU case, we constructed 1000 different STNUs, and determined whether they were dynamically, delay, or strongly controllable. Strong controllability acts as a more conservative version of delay controllability, labeling 21.4% of problem instances unsolvable where a valid policy does in fact exist (Table 4.2). In contrast, dynamic controllability ignores some of the constraints of the problem and in 43.1% of problem instances claims a viable strategy exists when one cannot in fact be guaranteed (Table 4.3). Delay controllability represents a significant improvement above and beyond existing controllability checks. Absent significant differences in the efficiency of these algorithms, there is no reason to prefer using strong or dynamic controllability to model communication delays in the resolution of temporally uncertain events. The rest of our results serve to explore this line of argumentation in detail and show that delay controllability checking performs well in practice (Figure 4-24).

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

Table 4.2: Delay vs. strong controllability results.

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

Table 4.3: Delay vs. dynamic controllability results.

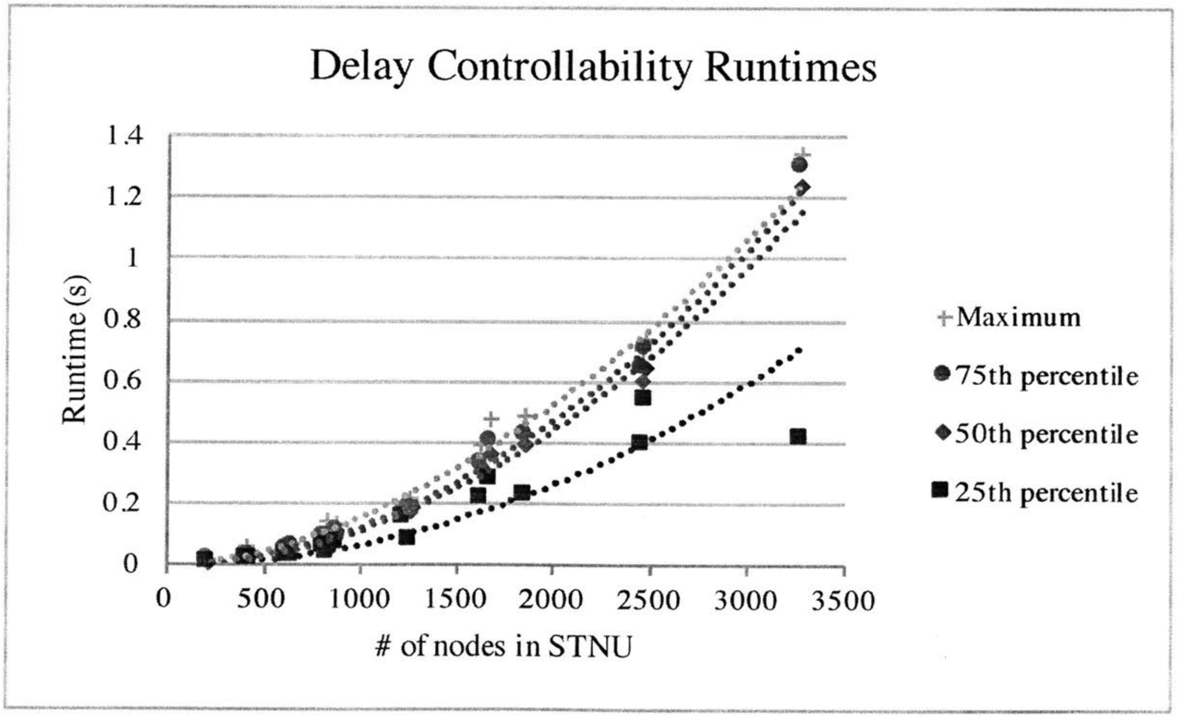


Figure 4-24: Runtime of delay controllability checkers on STNUs of different sizes. Shown are the 25th, 50th, 75th percentile, and maximum runtimes for STNUs of each size.

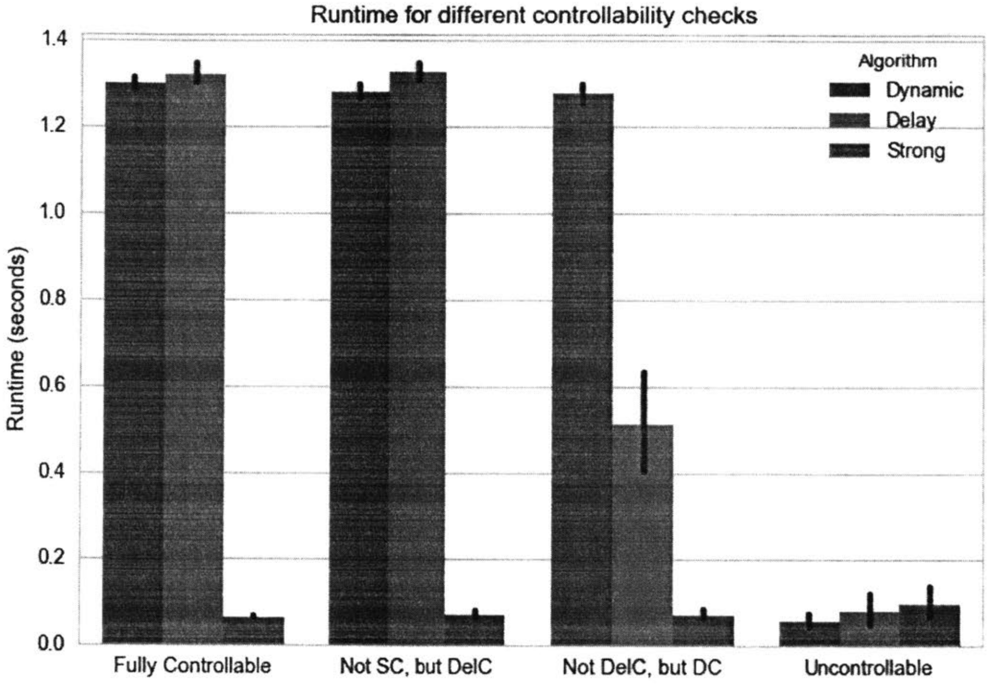


Figure 4-25: Runtimes of dynamic, delay, and strong controllability checkers on large STNUs. The runtimes are split based on the controllability of the network. DelC stands for delay controllability, DC stands for dynamic controllability, and SC stands for strong controllability.

In turning to runtime analysis, we shift our focus to the AUV experiments. 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 sufficiently fast for use in practice, taking under 1.5 seconds on the biggest networks we experimented with (see Figure 4-24). Most actual temporal network models have no more than a few hundred nodes, meaning that for most purposes delay controllability checking can be used as an efficient subroutine in more complex planning systems. For reference, AUV experiments conducted jointly with MERS and WHOI in November 2019 off the coast of Santorini involved no more than three vehicles operating simultaneously, each with fewer than 30 activities. The AUV in use was compute-limited but was able to run a fully capable delay controllability checker and dispatcher well within the sub-second budget afforded to it.

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 at our maximum problem size of 40 vehicles and 40 activities per vehicle, observing the runtime of dynamic, delay, and strong controllability algorithms (Figure 4-25). 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. We do so in order to understand the difference in performance of these algorithms at boundary conditions.

The resulting experiments demonstrated that, as expected, strong controllability checking proceeds significantly faster than delay controllability checking, and that there is a slight difference between dynamic and delay controllability checking, but that difference is not statistically significant. In our problem instances, strong controllability checks complete significantly faster than other controllability checks, taking under 0.1 seconds (see Figure 4-25), making it the fastest of all checks. When delay controllability and dynamic controllability checks return the same answer, whether indicating success or failure, delay controllability checking incurs a slight overhead, though the difference is not statistically significant; when both say the problem is controllable they take 1.3 seconds, whereas when they both see that the network is uncontrollable, they take under 0.1 seconds. When the two results do differ, delay controllability returns a result much more quickly, taking on average 0.5 seconds (see Figure 4-25). This matches intuitions from prior work, which empirically demonstrated that networks that are uncontrollable return faster from controllability checking procedures than networks that are controllable [6]; in instances where delay and dynamic controllability disagree, the given network would be dynamically but not delay controllable. This is important for planning, not only because correctness must be guaranteed in the presence of communication delays, but because most of the time spent by planners is in refuting infeasible candidate plans.

4.6.3 Discussion

On their own, these results match existing expectations and 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.

The differences in performance across different situations encourage us to take a portfolio approach to solving controllability problems, where we first check strong controllability and, in the event that the network is uncontrollable, we then check delay controllability. Strong controllability checks are overly conservative but, especially on large networks, finish much faster than delay controllability checks. In the event that an STNU is strongly controllable, checking strong controllability first saves a significant amount of time, whereas in the event that the network is not strongly controllable but is delay controllable, we only add a small overhead. (Note that the exact amount of overhead depends largely on the temporal network's size.).

Adopting this approach, however, depends heavily on the statistical distributions associated with the underlying temporal networks. If networks tend to often be strongly controllable, then checking strong controllability first has the potential to incur dramatic savings. In contrast, if the input networks tend not to be strongly controllable, regardless of whether they are delay controllable, the savings afforded by strong controllability checks are likely irrelevant as most strong controllability checks are immediately followed by a (comparatively) expensive delay controllability check.

4.7 Conclusion

In this chapter, 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 expressiveness beyond dynamic and strong controllability while still being fast enough to use in practice.

In offering a set of algorithms for efficient evaluation of delay controllability, we have now constructed a communication model that can be efficiently used by real-time executives when evaluating the feasibility of multi-agent problems. This work provides the backbone for the work in the remaining chapters by providing a medium for discussing delayed communication in multi-agent execution and is used when constructing communication strategies and handling uncertain and noisy communication.

Chapter 5

Determining Communication Strategies during Plan Execution

When executing multi-agent temporal plans, individual agents are uncertain about the behavior of other agents and must communicate in order to resolve that ambiguity. While these agents might have the ability to promptly deliver updates, there may be reasons why delivering immediate updates is expensive. For example, an autonomous underwater vehicle may have equipment on-board to immediately transfer information to any other agent, but powering it up may use precious battery power, needed for the rest of its mission. It may make sense to use a slower and less expensive means of communicating about its actions or to omit relaying information about that action altogether.

In Chapter 3, we defined the Communication Cost Minimization Problem (CCMP), whose aim is to establish a set of communication windows that guarantee plan success, while minimizing the associated cost of communication. To verify whether any particular set of communication windows is feasible, we can evaluate the windows in the context of a temporal network using delay controllability (see Chapter 4). However, two open questions remain. First is the question of how to effectively pick communication windows when there are costs associated with communicating at particular times and how to bound the communication costs associated with a given plan. The second related question is how to adjust the provided communication windows when communication deviates from the expected plan during execution. It is important to note that our approach is agnostic as to the particulars of the input delay-cost function, as long as the delay-cost function is admissible (i.e. it is not more costly to learn information later). This chapter focuses on how to derive solutions to CCMPs and offers three main contributions.

First, we provide an algorithm for deriving delay controllability conflicts, which will be used to guide our search for solutions to CCMPs. We augment existing delay controllability detection algorithms to output a conflict whenever a temporal plan is uncontrollable due to overly delayed communication. The returned conflict provides a disjunction of inequality constraints with the requirement that at least one must be satisfied in order to make the overall problem controllable. This is essential, as these conflicts allow us to employ conflict-directed search techniques to find potential solutions to CCMPs over an otherwise continuous and unbounded space [60]. Our work on delay controllability conflict extraction matches the best-known performance of dynamic controllability conflict extraction techniques [6].

Second, we explore three different variants of conflict-directed search for CCMPs, one guaranteed to output the lowest-cost delay function that still renders the input STNU delay controllable and two whose outputs may not be optimal. We analyze guarantees and performance of each of them. We show that, for certain STNUs, the sub-optimal algorithms can provide at best a polynomial approximation of the true optimal cost, but that in practice they provide a reasonable approximation, while performing faster in practice.

Third, we provide a series of algorithms for adjusting communication windows online when communication deviates from expected execution. In situations where planned communication is inherently unstable, causing communication to drop-out and reconnect sporadically, agents should be able to adjust their planned communication when necessary to adapt to failure as well as when adjustment can opportunistically improve the outcome of a plan. Our approach in constructing these algorithms involves maintaining an ever-approaching temporal horizon, after communications are interrupted, that represents the latest point in time that communication can be restored, in order for the remaining execution to satisfy all constraints. The online scheduler is then able to optimistically keep execution alive until communication is restored or the horizon is exceeded. Our temporal horizon maintenance makes heavy use of the delay controllability conflict extraction techniques developed in service of the offline CCMP solver to find the optimal horizon and maintains a store of those conflicts to efficiently relax our temporal horizon when communication is restored.

5.1 Approach

In this chapter, we are interested in solving the problem of determining when agents need to communicate about their problems. We use two different strategies to approach this problem, an offline one and an online one.

The offline problem constructs a set of deadlines for communication that guarantee success and are optimal with respect to a cost associated with communication. Formally, the offline problem can be solved directly, by generating valid solutions to a CCMP. A CCMP takes as input an STNU S and an admissible delay-cost function C, which assigns a real-value cost to each possible delay function γ. A solution to a CCMP is a delay function γ, such that S is delay controllable with respect to γ; an optimal solution is the delay function γ that has minimum cost $C(\gamma)$ across all solution to the CCMP.

We say that a delay function γ that is a solution for a CCMP represents a communication window for the original STNU. In other words, the delay function γ describes for each contingent event x_e, the amount of time that an agent may wait before communicating the timing of that action. As such, we use communication window and delay function interchangeably when describing algorithms for solving CCMPs.

Searching for an optimal-cost solution to a CCMP can be expensive, as the search space is continuous and unbounded; the solution to a CCMP is a delay function that outputs a real-valued delay for each contingent event. To handle this difficulty, the algorithm should learn from infeasible solutions to guide our search towards an optimal result. To do so, we rely on *conflicts*, which serve as certificates explaining why our original network is uncontrollable.

We break our original problem into a master and sub-problem. The goal of the master problem is to generate a candidate solution, in other words a delay function, that resolves all known conflicts and admits a low cost under the input cost function. The sub-problem evaluates the delay controllability of the input STNU with respect to the master problem's chosen delay function and outputs a conflict on failure.

This chapter presents the machinery for solving the master and sub-problem. We start with the sub-problem present a conflict extraction subroutine that can be used in conjunction with a delay controllability checking algorithm to extract conflicts efficiently. We then present a series of different conflict-directed search algorithms that are used in subroutines by the master problem in order to generate new candidates. Together these two systems work in concert to solve the original CCMP.

This fundamental work on handling the offline approach is instrumental for solving the online case. In the online version of the problem, agents may deviate from the prescribed deadlines for communication. An effective online algorithm should be able to reactively adapt to missed deadlines, but should also be opportunistic when communication happens sooner than expected.

The online version of the problem can be trivially reduced to the offline problem by recomputing optimal strategy after all events and communications. However, each re-computation involves inputs that are virtually identical. In this chapter, we also discuss how to exploit these similarities in cases where communication disruption comes in the form of unexpected outages. In service of this problem, we present a series of algorithms that are used to maintain temporal horizons in the presence of unexpected changes in communication. In response to these changes, execution is maintained for as long as possible and can optimistically recover when other approaches may have preemptively halted

5.2 Conflict Extraction

We start by introducing an algorithm that explains how to extract delay controllability conflicts. This routine is crucial both for offline and online approaches to determining when agents must communicate, but we will discuss this algorithm initially in the context of solving the sub-problem of the offline method.

The problem of finding an offline solution to a CCMP is partially that of understanding which set of delay functions can be used to ensure that the input STNU is delay controllable (the other part is finding a low-cost delay function). In order to find this set of delay functions, we can use conflict-directed search (as part of the master problem) to understand what properties of delay functions would make the original STNU uncontrollable. As such, we need a way to extract conflict related to delay controllability.

In the case of delay controllability, we know that the presence of a semi-reducible cycle in our STNU represents a conflict, and from it, we can extract reasons why a particular choice of delay function was uncontrollable. Correspondingly, it makes sense to modify the algorithm used by our sub-problem for checking delay controllability, changing it into one that checks delay controllability and extracts a conflict if the STNU is not delay controllable. Our conflict-extraction algorithm builds on top of the original delay controllability algorithm [3] through the maintenance of some additional state and is based on the original work of conflict extraction and resolution by [21].

The original delay controllability algorithm (Algorithm 3) works by invoking a variant of Dijkstra's algorithm from each negative weight edge. The calls to Dijkstra's algorithm are responsible for finding the shortest semi-reducible paths from each node to all others in the graph. The subroutine is recursively invoked any time another negative edge is found, and when an infinite recursion is detected, we know that we have found a cycle composed of at least one and possibly many semi-reducible negative paths. More detail on the correctness of the original algorithm can be found in Chapter 4.

```
Input: Labeled distance graph, $G = \langle V, E \rangle$;
delay function $\gamma$
Output: Whether the STNU derived from the distance graph is delay
        controllable and if not, the edges embodying the conflict
Initialization:
1 $negNodes \leftarrow$ the set of all vertices with incoming negative edges;
2 $novel \leftarrow []$; list of newly added edges;
3 $preds \leftarrow \{\}$; mapping of function call to predecessors;
DELAYCONTROLLABLE?:
4 for $v \in negNodes$ do
5     $cycleFree?, edges \leftarrow$DELAYCONFLICTDIJKSTRA($G$, $\gamma$, $v$, $preds$, $novel$,
        $[v]$, $negNodes$);
6     if $!cycleFree?$ then
7         return $false$, EXTRACTCONFLICTS($edges$, $novel$, $preds$)
8 return $true, \emptyset$
```

Algorithm 3: Delay Controllability algorithm that reports conflicts

To efficiently extract delay controllability conflicts, we augment Algorithm 3 to return the detected conflict. Lines 16-21 of Algorithm 2 are where we assemble the edges that compose the semi-reducible negative cycle. Whenever a recursive call to DELAYCONFLICTDIJKSTRA returns false, we know that at some point in the call stack, we discovered a semi-reducible negative cycle. However, the entire chain of edges is not necessarily part of the cycle. We use the third return value of DELAYCONFLICTDIJKSTRA to specify one node that is known to be part of the cycle. At line 20, we augment the list of edges that compose the negative cycle, and at line 21, we signal that we have fully specified a semi-reducible negative cycle because we have returned to a node we have already visited.

A key property of this new algorithm is that it preserves the algorithm's $O(n^3)$ runtime, as maintaining the additional data structures only incurs a constant overhead. Each call to EXTRACTEDGEPATH adds at most n edges to our list, and EXTRACTEDGEPATH is called at most once per call to DELAYCONFLICTDIJKSTRA. Because DELAYCONFLICTDIJKSTRA is called at most once per node, it adds an additional overhead of $O(n^2)$, which is dominated by the normal runtime of the algorithm.

Finally, the call to EXTRACTCONFLICTS in line 7 of Algorithm 3 takes the list of edges composing the cycle and replaces any newly added edges with the original

```
Input: Labeled distance graph $G = \langle V, E \rangle$, delay function $\gamma$, start node $s$, list of
       predecessor edges $preds$, list of new edges, $callStack$, and $negNodes$
Output: Whether the current walk is cycle-free, and the edges composing a
        semi-reducible negative cycle
Initialization:
$Q \leftarrow PriorityQueue()$;
$labelDist \leftarrow \{s : \langle 0, \emptyset \rangle\}$; $unlabDist \leftarrow \{s : \langle 0, \emptyset \rangle\}$;
for $e \in s.incomingEdges()$ $if$ $e.weight < 0$ $and$ $!e.lowerCase()$ do
    $Q.add(\langle e.from, e.label \rangle, e.weight)$;
    $(e.label == \emptyset\ ?\ unlabDist : labelDist)[e.from] \leftarrow \langle e.weight, e \rangle$
DelayConflictDijkstra:
if $s \in callStack[1 : end]$ then
    return $false, \emptyset, s$;
$preds[s] \leftarrow \langle labelDist, unlabelDist \rangle$;
while $Q.size() > 0$ do
    $v, label, weight \leftarrow Q.pop()$;
    if $weight \geq 0$ then
        $G.add(\langle v, s, weight \rangle)$;
        $novel.add(\langle v, s, weight \rangle)$;
        $continue$;
    if $v \in negNodes$ then
        $newStack \leftarrow [v].concat(callStack)$;
        $result, edges, end \leftarrow$ DelayConflictDijkstra($G$, $\gamma$, $v$, $preds$,
          $novel$, $newStack$, $negNodes$);
        if $!result$ then
            if $end \neq \emptyset$ then
                $edges.add($ExtractEdgePath($s$, $v$, $labelDist$, $unlabelDist$)$)$;
            $end \leftarrow (end == s)\ ?\ \emptyset : end$;
            return $false, edges, end$;
    for $e \in v.incomingEdges()$ $where$ $e.weight \geq 0$ $and$ $(!e.isLowerCase()$ $or$
      $e.label \neq label)$ do
        $w \leftarrow e.weight + weight$;
        $l \leftarrow (label \neq \emptyset$ and $w > -\gamma(label))\ ?\ \emptyset : label$;
        $dist \leftarrow l \neq \emptyset\ ?\ labelDist : unlabDist$;
        if $Q.addOrDecKey(\langle e.from, l \rangle, w)$ then
            $dist[e.from] \leftarrow \langle w, e \rangle$;
        $lower \leftarrow (e.from).incomingLowerEdge()$;
        if $lower \neq null$ $and$ $e.weight < \gamma(lower.label)$ $and$
          $Q.addOrDecKey(\langle lower.from, l \rangle, w + lower.weight)$ then
            $dist[lower.from] \leftarrow \langle w + lower.weight, lower \rangle$;
$negNodes.remove(s)$;
return $true, \emptyset, \emptyset$;
```

Algorithm 4: Function DelayConflictDijkstra

edges that were used to derive them. The resulting output is our conflict.

5.2.1 Resolving Conflicts

Now that we have a way of extracting conflicts when our STNU is uncontrollable, we need a way to reason about and resolve them. In this context, resolving a conflict involves specifying a set of constraints that must hold for our chosen delay function in order for the input STNU to be delay controllable. Since delay controllability conflicts manifest as semi-reducible negative cycles, resolving these conflicts means we need to adjust our delay function in a way that prevents the semi-reducible negative cycle from forming.

Given a semi-reducible negative cycle, there are two ways to eliminate it: make the cycle non-negative or make it non-semi-reducible. Since our communication windows and choice of γ have no impact on the length of edges in the STNU's labeled distance graph representation, modifying our communication windows will not affect the weight of the cycle. Hence, we should instead focus our attention on how to modify γ to eliminate the property of semi-reducibility.

To modify γ to eliminate the property of semi-reducibility, we need to ensure that our modification prevents the application of some reduction rule that was used in the creation of the original semi-reducible negative cycle. There are three reduction rules that involve γ: the label removal rule, the lower-case reduction rule, and the cross-case reduction rule (see Table 4.1). We need only focus on the latter two in order to resolve all conflicts.

Lemma 5.1. *Adjusting γ to eliminate a label removal reduction is not sufficient to eliminate a semi-reducible negative cycle.*

Proof. With the reduction rules we provided, we know that all generated upper-case edges which share a label end at the same node, which is the starting node of the corresponding lower-case edge. If a label removal was needed to allow a lower-case reduction by the corresponding lower-case edge, the lower-case edge and upper-case edge whose label is being removed would form a cycle since the original labeled edges share that endpoint. This gives us two possibilities.

In the first, the resulting cycle is non-negative. If this were the case, we could excise the sub-cycle and still be left with a semi-reducible negative cycle. This means that adjusting γ to try to eliminate the label removal reduction is strictly less useful than adjusting it to eliminate other reductions.

In the second, the generated cycle is negative. Assume that the original contingent edge is of the form $A \overset{[u,v]}{\Longrightarrow} B$ and that we can reduce the upper-case edge to some form $A \xrightarrow{B:x} C$ where $x > -u$ in order to apply the label removal reduction. Notice, however, that this situation is impossible. In order for the label removal to be necessary to generate the semi-reducible negative cycle, every prefix of the path between the lower-case edge and upper-case edge needs to have weight at least $\gamma(B)$ in order to preclude a lower-case reduction. This means that the total weight of the cycle is at least $u + \gamma(B) + x$. But because $x > -u$, we have that the total weight of the cycle is greater than 0. Thus, we do not need to worry about adjusting γ to preclude label removals.

□

Now, we explain how to change γ to resolve delay controllability conflicts and how we have a guarantee that iterative resolutions will eventually lead us to a valid solution.

Theorem 5.2. *If an STNU is controllable when $\gamma = 0$ (dynamically controllable), then if the STNU has a delay controllability conflict for any particular choice of γ, we can always adjust γ to eliminate a lower-case or cross-case reduction.*

Proof. A semi-reducible negative cycle is one where we can apply edge reductions to eliminate lower-case edges. In particular, the lower-case and cross-case reduction rules are the rules directly responsible for eliminating those edges, and by Lemma 5.1, we know it is safe to ignore label removal reductions.

For every lower-case edge from our conflict's semi-reducible negative cycle, we use the following approach for invalidating the reduction and thus stopping the formation of the semi-reducible negative cycle. From a lower-case edge with label b, we find the shortest subpath of the cycle that immediately follows the lower-case edge such that its total weight is less than $\gamma(B)$. We know such a subpath exists because all lower-case edges have non-negative weight and the total weight of the cycle is negative. If the weight of the subpath is non-negative, we adjust our $\gamma(B)$ to be equal to its weight.

If we cannot adjust any value of γ because all of the successive subpaths are negative, then we have a contradiction. This same semi-reducible negative cycle would still be present when $\gamma = 0$, and the original STNU would not be dynamically controllable.

□

5.3 Minimum-Cost Communication

With an efficient way to extract delay controllability conflicts and solve our subproblem, we can now focus on identifying low-cost and ultimately minimum-cost values of γ which yield delay controllable networks in service of solving the master problem. In this section, we present three solutions that are all based on a form of conflict-directed search. The first two have no optimality guarantees but are fast in practice, and the third is a conflict-directed best-first search that is guaranteed to yield an optimal value of γ.

5.3.1 Conflict-Directed Search

Our initial approach at finding a feasible set of communication windows (Algorithm 5) uses conflicts to iteratively refine its choice of delay function γ. Before we proceed with a potentially costly search process, we first check at line 1 to see whether the original STNU is dynamically controllable (or whether it is delay controllable with respect to $\gamma = 0$). Since decreasing observation delays always preserves controllability, we know that if the STNU is not dynamically controllable, we have no chance of finding a suitable delay function and can safely avoid the search process.

Once we have a guarantee that there does exist some value of γ which makes

```
Input: Labeled distance graph, $G = \langle V, E \rangle$ for STNU;
Output: A valid delay function $\gamma$ or $\emptyset$ if one does not exist;
GREEDYCOMMCOST:
1 if !DELAYCONTROLLABLE?$(G, \gamma(\_) = 0)$ then
2     return $\emptyset$;
3 $candidate \leftarrow \gamma(\_) = \infty$;
4 $controllable, conflict \leftarrow$ DELAYCONTROLLABLE?$(G, candidate)$;
5 while $!controllable$ do
6     $candidate \leftarrow candidate.pickResolution(conflict)$;
7     $controllable, conflict \leftarrow$ DELAYCONTROLLABLE?$(G, candidate)$;
8 return $candidate$;
```

Algorithm 5: GREEDYCOMMCOST, an algorithm that performs different variants of greedy search to find a valid delay function γ for an input STNU that is delay controllable.

the STNU delay controllable, we can begin our search for an optimal delay function. Each conflict returned from the delay controllability check (lines 4, 7) represents a disjunction of modifications we could make to our delay function to eliminate this particular conflict.

A brief walkthrough of the operation of Algorithm 5 is shown in Figures 5-1 and 5-2. The algorithm starts by checking that the input STNU (Figure 5-1a) is dynamically controllable. Once satisfied that it is, it constructs a delay function where $\gamma(x_c) = \infty$ for all x_c and checks delay controllability.

After our first check, we see that the network is not delay controllable with respect to γ and receive back a conflict (see edges in red in Figure 5-1b). The only way we can resolve the conflict in this context is by changing the input delay function in a way that eliminates the derived semi-reducible negative cycle. We know that two lower-case reductions were applied to yield this cycle. The first involves $A \xrightarrow{b:0} B$ and $B \xrightarrow{0} C$. The second involves $C \xrightarrow{d:0} D$ and $D \xrightarrow{1} E$. The conflict can then be resolved if either $\gamma(B) \leq 0$ or $\gamma(D) \leq 1$.

The specific choice of how the conflict is resolved depends on the resolution method and the associated delay-cost function, but for the sake of the walkthrough, we can assume that γ is modified by setting $\gamma(D) = 1$ (see Figure 5-2a). Checking for delay controllability yields another conflict, and though the edges involved are the same

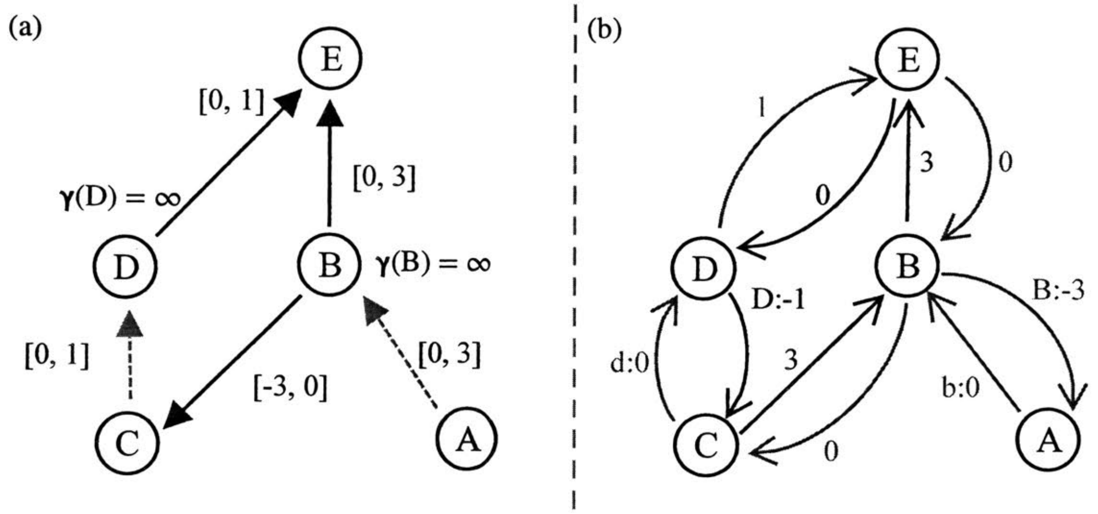


Figure 5-1: (a) Our initial input STNU. The initial delay controllability check is performed assuming that the two contingent events, B and D are completely unobserved. (b) The STNU is not delay controllable with respect to the given γ and the delay controllability conflict is shown in red.

(see Figure 5-2b), the potential set of resolutions is not.

To resolve the conflict we must eliminate a lower-case or cross-case reduction used to create the semi-reducible negative cycle. The first lower-case reduction again uses edges $A \xrightarrow{b:0} B$ and $B \xrightarrow{0} C$. The other reduction is this time a cross-case reduction involving $C \xrightarrow{d:0} D$ and $D \xrightarrow{B:-2} A$, where $D \xrightarrow{B:-2} A$ is generated from edges $D \xrightarrow{1} E$, $E \xrightarrow{0} B$, and $B \xrightarrow{B:-3} A$. It is worth noting that adjusting γ cannot eliminate the cross-case reduction since we cannot introduce a negative delay in observation. As a result, the only remaining way to resolve the conflict is to set $\gamma(B) = 0$, and when checking for delay controllability we find that the STNU is in fact delay controllable with respect to the newly derived γ.

In the walkthrough, we were intentionally vague about how we picked between the two possible resolutions for our conflict. Here we introduce two such methods for doing so. The first two approaches that we introduce and evaluate, blind search and lowest-cost-resolution search (LCRS), employ different conflict resolution strategies to find an approach that works and are represented by different implementations of the *pickResolution* function in line 6 of Algorithm 5. Instead of keeping track of all possible branches, we choose a single disjunct to resolve and continue checking for

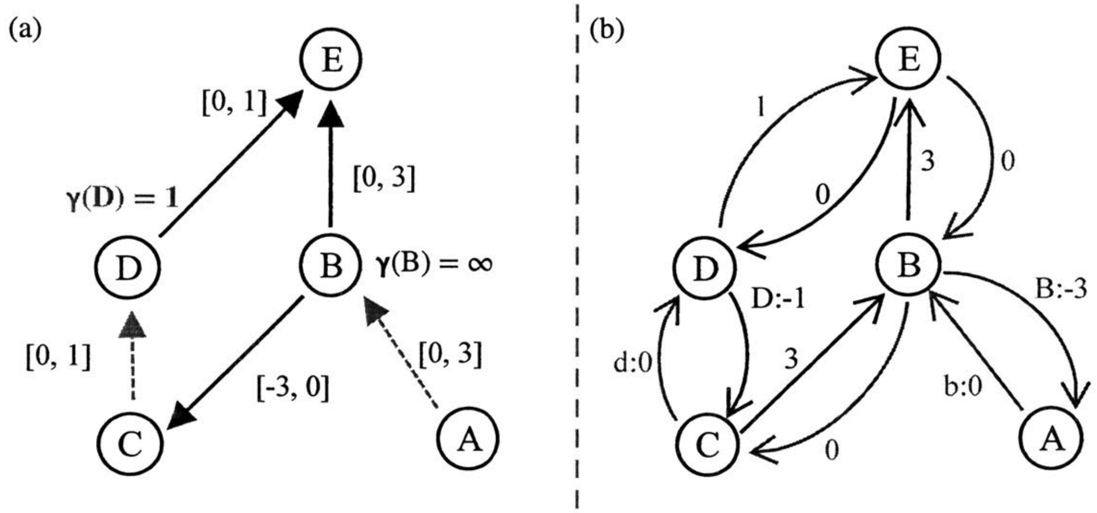


Figure 5-2: (a) After one iteration, γ is updated to rectify the original conflict and we set $\gamma(D) = 1$. (b) We generate the same set of edges (in red) for the conflict, but the choices we have to resolve it are slightly different because of the input γ.

delay controllability. Note that any potential solution must resolve at least one of the disjuncts in every available conflict, but doing so is not sufficient to guarantee that the solution yields a delay controllable STNU. If our choice was insufficient, subsequent controllability checks would yield additional conflicts.

In the case of blind search, we non-deterministically commit to any of the possible conflict resolutions, and in the case of LCRS, we commit to the conflict resolution with lowest possible cost. In expectation, blind search is quicker to pick a conflict resolution, but LCRS may find solutions that are overall lower in cost. While neither approach is optimal, both approaches are guaranteed to eventually find a satisfying delay function γ that yields a controllable STNU since they will eventually resolve all conflicts.

5.3.2 Conflict-Directed Best-First Search

While blind search and LCRS are appealingly simple, that they are not guaranteed to be optimal is cause for concern. In fact, we are unable to provide guarantees that our search procedures are within a constant factor approximation of optimal. In extreme instances, these search processes can yield results that are polynomially worse than

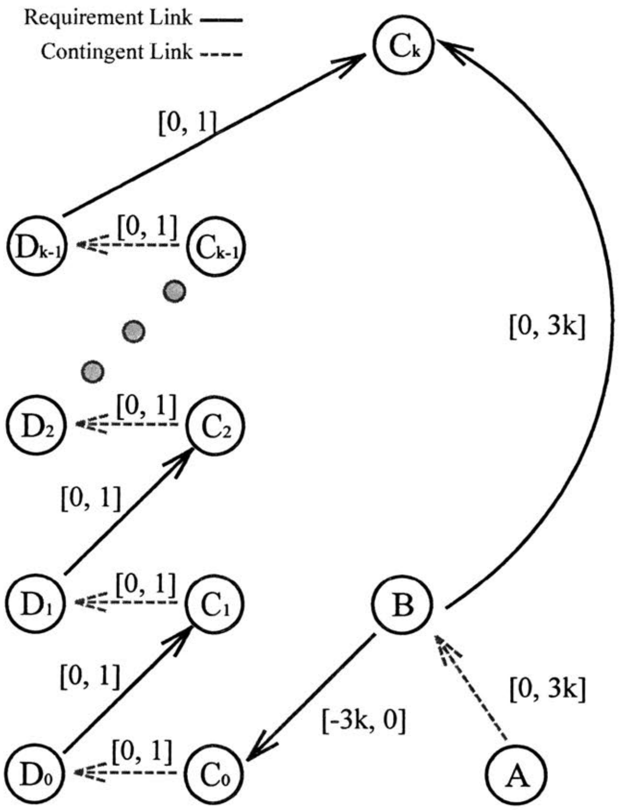


Figure 5-3: This k-adversarial STNU is dynamically controllable but is not delay controllable if $\gamma = \infty$. Requiring that $\gamma(B) = 0$ is necessary and sufficient to make the STNU delay controllable.

optimal.

Theorem 5.3. *Blind search and LCRS are not guaranteed to yield results within a constant factor of optimal.*

Proof. Consider the STNU presented in Figure 5-3, which we call a k-adversarial STNU, and imagine that we want to find a minimum-cost delay function γ making it controllable. For simplicity, assume that our cost function is $C(\gamma) = \sum_{x_c} \frac{1}{1+\gamma(x_c)}$. When called on this STNU, DELAYCONTROLLABLE? identifies a semi-reducible negative cycle that includes $A \Rightarrow B$ and $C_i \Rightarrow D_i$ for all $0 \leq i < k$. The corresponding resolution to that conflict requires that either $\gamma(B) = 0$ or for some $0 \leq i < k$, $\gamma(D_i) = 1$.

With a blind approach, we pick random disjuncts to satisfy until the STNU is controllable. However, each choice to relax $\gamma(D_i)$ makes no overall progress towards the controllability of the STNU. Only when $\gamma(B) = 0$ does the STNU become controllable. In expectation, $\frac{k}{2}$ relaxations happen before blind search relaxes $\gamma(B)$. Since each relaxation of $\gamma(D_i)$ to 1 incurs a cost of $\frac{1}{2}$ and relaxing $\gamma(B)$ to 0 incurs a cost of 1, in expectation blind search incurs a cost of $1 + \frac{k}{4}$.

With LCRS, the results are even worse. LCRS always elects to resolve the conflict by letting some $\gamma(D_i) = 1$ since this incurs a cost of $\frac{1}{2}$ whereas letting $\gamma(B) = 0$ incurs a cost of 1. As such, the greedy approach updates $\gamma(D_i)$ for all k such contingent constraints before updating $\gamma(B)$. On this STNU, the greedy approach incurs a cost of $1 + \frac{k}{2}$, whereas the optimal approach has a cost of just 1. These results motivate our interest in developing an optimal algorithm.

□

```
Input: Labeled distance graph, $G = \langle V, E \rangle$ for STNU, and a cost function $C$;
Output: A delay function $\gamma$ that is of minimal cost or $\emptyset$ if one does not exist;
Initialization:
1  $queue \leftarrow []$ // queue of candidate $\gamma$ functions;
MinCommCost:
2  if !DelayControllable?($G, \gamma(\_) = 0$) then
3      return $\emptyset$;
4  $queue.append(\gamma(\_) = \infty)$;
5  $\gamma \leftarrow queue.pop()$;
6  $controllable, conflict \leftarrow$ DelayControllable?($G$, $\gamma$);
7  while !controllable do
8      $queue.add(\gamma.allResolvedConflicts(conflict))$;
9      $queue.sortBy(C)$;
10     $\gamma \leftarrow queue.pop()$;
11     $controllable, conflict \leftarrow$ DelayControllable?($G$, $\gamma$);
12 return $\gamma$;
```

Algorithm 6: Algorithm that computes the minimum-cost delay function for a given STNU.

Our optimal algorithm for finding a minimal communication cost (Algorithm 6) is a form of conflict-directed best-first search (CDBFS) and works as follows [60]. Like with the greedy algorithm, we immediately check if the STNU is controllable with respect to $\gamma = 0$ or whether it is dynamically controllable (lines 2-3).

Once we know that a solution can be found, we now have to find the lowest-cost solution. Since we know that our cost function C is component-wise monotonically decreasing, we start with the lowest cost communication pattern possible, $\gamma = \infty$, and add it to our queue.

Every time we use a delay function that makes the given STNU uncontrollable and results in a conflict, we use that conflict to generate a set of delay functions to try later. For the value of γ we last checked, we derive modifications that each satisfy one of the provided disjuncts. We always choose the lowest cost value that satisfies the disjunct, which occurs when the delay for a contingent event is exactly equal to the conflict's upper-bound. By Theorem 5.2, we know that we will eventually reach a solution and because we are using best-first search, we know it will be optimal.

5.4 Handling Communication Outages

The algorithms that we have introduced to this point have been focused on solving offline CCMPs. In this section, we describe one particular uncertainty model for online CCMPs that is common in practice and show how we can adaptively react to that uncertainty to preserve our ongoing plan for as long as possible where other approaches might prematurely abort.

5.4.1 Execution Model

As inspiration for our model, we consider the deployment of multiple autonomous underwater vehicles (AUVs) and a typical plan that might be generated for a candidate scientific mission [53, 1]. In a typical mission, a series of AUVs work in concert with a primary crew responsible for picking up and dropping off the AUVs at specific locations and times. Each AUV is responsible for independently navigating to and surveying scientific locations, and across all actions there is some degree of temporal uncertainty.

In an idealized model, the primary crew can constantly monitor the progress of the AUVs with the use of a low-bandwidth acoustic modem. In reality though, the acoustic modem may at times be unreliable, and it may not be possible to constantly communicate with an AUV. At certain moments during execution, the AUV may surface to better localize its position and at these times we can guaranteeably communicate with the vehicle.

This unreliable medium of communication will form the basis of our main model. We assume that each AUV has some degree of temporal flexibility with which it can perform its actions and that while those choices are made autonomously, they are in general immediately visible to the primary crew responsible for coordination and eventual pickup. In instances where communication becomes unreliable, our goal is to robustly monitor execution, only discarding the plan when the growing uncertainty makes it impossible to continue with the current plan and accounting for the fact that we may eventually receive the information we need. Crucially, we assume that communication for individual AUVs are independent, meaning that if we learn about one event that we had previously missed, we have no guarantee that we learn about other events, and if we fail to learn about an event, it does not negatively affect other events.

5.4.2 Monitoring Execution

A robust execution monitoring system that functions under unreliable communication needs to be able to make decisions based on the information currently available and to maintain a *temporal plan horizon* which dictates how long the current plan is feasible. In this chapter, we will focus primarily on the problem of maintaining a temporal plan horizon, as many solutions exist for dispatching plans under temporal uncertainty [23, 35]. By augmenting an existing plan executive with our set of data structures, namely a list of delay controllability conflicts, a list of missed events mapping events to the earliest times at which they might have actually occurred, a delay function $\hat{\gamma}$, and a single real-valued variable h representing the temporal plan horizon, it becomes possible to make that executive robust to interruptions in communication.

At the beginning of execution, our data structures are initialized with an empty conflict list, an empty list of missed event times, some initial delay function $\hat{\gamma}$, which we use to represent the expected set of delays in communication assuming no interruptions, and a temporal horizon set to $h = \infty$, which represents the time by which communication must be restored in order for execution to have a chance of successfully completing. As plan execution progresses, we update these data structures to account for changes made to ongoing communication. In particular, the two types of events that can affect a temporal plan's horizon are communications going down, causing us to miss potential events, and communications being restored, causing us to learn about events that we had previously missed.

To handle these two cases, we introduce two callbacks that when added to an execution monitoring system are able to appropriately handle changes to the underlying communication state. The first, MISSEDEVENTCALLBACK (Algorithm 7), is triggered when communications are down and we detect that it is possible that we missed some event. Whenever communication goes down and an event is missed, we update our delay function $\hat{\gamma}$ to indicate that the event is unobservable (line 2) and derive a new *operating delay function* (line 3) which represents a delay function under the belief that communication will be restored by the horizon h. The remaining work is to update our temporal horizon now that another event has been missed.

To do so, MISSEDEVENTCALLBACK uses delay controllability conflicts to determine the maximum delay from the event that was just missed that can be tolerated while still guaranteeing that the system as a whole is controllable. Each delay controllability conflict is a disjunction of linear inequalities of the form $\gamma(x_c) \leq d_c$ with the guarantee that at least one of the inequalities must be satisfied in order for the overall network to be controllable. We use the conflict-extracting version of the delay controllability algorithm (Algorithm 3) to efficiently find these conflicts to guide our search for a satisfying $\hat{\gamma}$.

The primary control loop of MISSEDEVENTCALLBACK involves checking if the STNU is controllable with respect to some updated delay function, $\hat{\gamma}'$ (lines 4-9). If the STNU is deemed to be uncontrollable, REQUIREDHORIZONFORCONFLICT (Algorithm 9) uses the returned conflict to derive a new horizon from which we can construct a delay function which satisfies the conflict.

Despite the fact that a conflict has many possible disjuncts that can be satisfied and therefore many possible delay functions that would satisfy the conflict, REQUIREDHORIZONFORCONFLICT can satisfy all conditions with a single new temporal plan horizon. By maximizing the possible temporal plan horizon, we are preferentially choosing to solve the conflict in a way that both resolves the conflict and maximizes overall flexibility.

REQUIREDHORIZONFORCONFLICT works by returning the latest possible horizon that satisfies the conflict or by returning $\emptyset$ which signals that the conflict is already solved by the input delay function $\hat{\gamma}$. The algorithm starts by iterating over every inequality in the conflict of the form $\hat{\gamma}(x_c) \leq d_c$ (Algorithm 9, line 2). If we find that our original $\hat{\gamma}$ already satisfies the conflict, we can return $\emptyset$ immediately (Algorithm 9, lines 3-4). If not, we know we can satisfy the conflict by updating our delay function to have $\hat{\gamma}'(x_c) = d_c$. Doing so puts our time horizon at $t_m(x_c) + d_e$, where $t_m(x_c)$ represents the earliest time that we believe event x_c might have occurred in accordance with the observed communication failure. As a result, across all events x_c contained in the conflict, we want to resolve the conflict by picking the x_c that maximizes $t_m(x_c) + d_e$ (Algorithm 9, lines 6-8). Note that the state required to calculate t_m is updated using the *missedEventTimes* data structure (Algorithm 7, line 1; Algorithm 8, line 1).

As we continue to check for controllability in MISSEDEVENTCALLBACK and get updated horizons from REQUIREDHORIZONFORCONFLICT, we eventually converge to a delay function that is controllable. If there is no further change to existing communication channels, then so long as communication is restored by the time specified by the temporal plan horizon, execution is guaranteed to succeed. If communication is not restored by the temporal plan horizon, then some constraints will necessarily be violated. However, because it is possible for communication to be restored, we need to consider how to take that new information into account to appropriately adjust our temporal plan horizon.

The second callback needed to maintain an accurate temporal plan horizon, OBSERVEDEVENTCALLBACK (Algorithm 8), is called after communication is restored

```
Input: Missed event $e$; current time $t$; STNU $S$; current delay function $\hat{\gamma}$;
       $missedEventTimes$; current horizon $h$; $conflictList$
Output: A modified delay function $\hat{\gamma}$ and horizon $h$ representing the time at
        which successful execution can no longer be guaranteed
Initialization:
1  $missedEventTimes[e] \leftarrow t$;
2  $\hat{\gamma}[e] \leftarrow \infty$;
3  $\hat{\gamma}' \leftarrow$ DelaysGivenHorizon($\hat{\gamma}, h, missedEventTimes$);
MissedEventCallback:
4  $contr?, confl \leftarrow$ DelayControllable($S$, $\hat{\gamma}'$);
5  while not $contr?$ do
6      $conflictList.append(confl)$;
7      $h \leftarrow$ RequiredHorizonForConflict($confl, missedEventTimes, \hat{\gamma}'$);
8      $\hat{\gamma}' \leftarrow$ DelaysGivenHorizon($\hat{\gamma}, h, missedEventTimes$);
9      $contr?, confl \leftarrow$ DelayControllable($S$, $\hat{\gamma}'$);
10 return $\hat{\gamma}, h$;
```

Algorithm 7: Callback when an expected communication event is missed.

and we learn the true times that certain events occurred. The strategy of ObservedEventCallback closely resembles that of MissedEventCallback; after recording the newly observed event, the algorithm makes the optimistic assumption that we can guarantee controllability without observing any of the events that were lost due to communication outages. It then uses RequiredHorizonForConflict to tighten the temporal plan horizon.

Unlike in the approach of MissedEventCallback, we can avoid the expensive calls to DelayControllability by reusing the conflicts that were derived from previous callback invocations (see Algorithm 7, line 1). Under our model, learning about an event after communication is restored is equivalent to reducing the delay in its observation, and reducing delay in observation never introduces new conflicts.

We iterate through all conflicts, checking if the conflicts are resolved by the new information (Algorithm 8, lines 5-6), and for those that do not, we take the minimum required horizon across all conflicts as our final horizon (lines 7-8). By picking the minimum horizon across all conflicts, we guarantee that our choice of horizon

Together, these two callbacks allow us to actively maintain our temporal plan horizon, allowing us to maximize the time before re-planning becomes necessary.

```
Input: Missed event $e$; current time $\tau$; event time $t_e$; STNU $S$; current delay
       function $\hat{\gamma}$; current horizon $h$; $missedEventTimes$; $conflictList$
Output: A modified delay function $\hat{\gamma}$ and horizon $h$ representing the time at
        which successful execution can no longer be guaranteed
Initialization:
$missedEventTimes.remove(e)$;
$\hat{\gamma}[e] \leftarrow \tau - t_e$;
ObservedEventCallback:
for $confl \in conflictList$ do
    $h' \leftarrow$ RequiredHorizonForConflict($confl, missedEventTimes, \hat{\gamma}$);
    if $h' == \emptyset$ then
        $conflictList.remove(confl)$;
    else if $h' < h$ then
        $h \leftarrow h'$;
return $\hat{\gamma}, h$;
```

Algorithm 8: Callback when communication is restored and we learn about a past event.

```
Input: conflict $c$; existing delay function $\hat{\gamma}$; $missedEventTimes$
Output: A new horizon representing the time at which communication must be
        restored in order for the conflict to be resolved
Initialization:
$h \leftarrow \emptyset$;
RequiredHorizonForConflict:
for $(event, d_e) \in conflict$ do
    if $\hat{\gamma}[event] \leq d_e$ then
        return $\emptyset$;
    if $\exists t_m : (event, t_m) \in missedEventTimes$ then
        $newHorizon \leftarrow t_m + d_e$;
        if $h == \emptyset$ or $h < newHorizon$ then
            $h \leftarrow newHorizon$;
return $h$;
```

Algorithm 9: Derive a new horizon given a delay controllability conflict.

5.5 Experimental Results

In this section, we provide empirical evaluations of our algorithms for solving CCMPs, showing the empirical efficacy of our suboptimal search algorithms for pre-computing communication strategies. In particular, we are interested in evaluating the relative speed and performance of the different approaches for solving the master problem and to determine which of the three approaches are best-suited for use in practice.

Qualitatively, the search algorithms we have presented so far have very different properties. Blind search and LCRS have much lower overhead but have no guarantees of optimality. In contrast, CDBFS is guaranteed to output a delay function that is optimal with respect to the inputted cost function but can be much slower. In this section, we present our empirical analysis as a way to better characterize exactly how different these approaches are and when it might be preferable to use one over the other. Across all examples, we see that LCRS and blind search tend to run much faster than CDBFS, and in more typical instances, we see that the solutions returned by LCRS are near optimal.

5.5.1 Experiment Setup

Our experimental analysis will compare the performance of our three different search algorithms, both in terms of speed and in terms of the quality of the solutions they return

We start by examining the performance of our algorithms on k-adversarial STNUs. We performed 50 trials each with parameters $k = 10, 20, 30, 40, 50$, and for all experiments, we assumed a cost function of $C(\gamma) = \sum\limits_{x_c} \frac{1}{1+\gamma(x_c)}$ as is typical in the k-adversarial STNU setup. This analysis allows us to examine behavior in the most difficult scenarios. While we expect the quality of the suboptimal algorithms, blind search and LCRS, to match the theoretical results which indicate that they perform polynomially worse than CDBFS, we also care to understand what the runtime tradeoffs across all three algorithms look like.

k-adversarial STNUs present a significant challenge to all of our algorithms, but most STNUs in practice do not look like our k-adversarial ones. In order to validate whether the trends observed over k-adversarial STNUs hold more generally, we want to run our experiments for randomly generated STNUs as well.

Our random STNUs are composed of k independent contingent constraints which each have a lower-bound of 0 and an integer upper-bound uniformly chosen between 1 and 4. For each pair of endpoints between distinct contingent constraints, we add a requirement constraint between the two with probability $\frac{1}{4k}$. Since there are a total of $4k(k-1)$ constraints that could be drawn between them, this gives us STNUs that have in expectation $k-1$ requirement constraints between the k contingent constraints. Each requirement constraint has a lower-bound of 0 and an integer upper-bound chosen uniformly between 1 and 4. For our analysis, we ensure that our 50 trials are selected from the set of random STNUs that are dynamically controllable but not strongly controllable. In instances where the STNU is either not dynamically controllable or is already strongly controllable, the algorithms either reject or accept immediately, respectively, making comparisons across the algorithms uninteresting.

5.5.2 Results

We start by considering the results of our algorithms when run on k-adversarial STNUs. As expected, wee see that LCRS always outputs a solution with cost $1 + \frac{k}{2}$, blind search outputs a cost that converged in expectation to $1 + \frac{k}{4}$, and CDBFS always outputs a unit cost solution. Thus, from an optimality perspective, we know that for k-adversarial STNUs, blind search and LCRS perform polynomially worse than CDBFS.

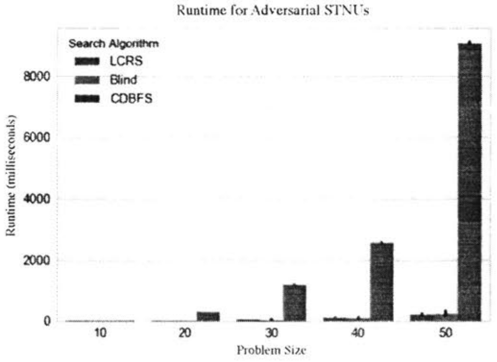


Figure 5-4: The runtimes of the solutions outputted by the search algorithms when run on k-adversarial STNUs.

However as we scale up the problem size, we see that the difference in runtime performance between the suboptimal approaches and CDBFS starts to grow (Figure 5-4). For each of the different problem sizes, we find that CDBFS has a significantly slower runtime than the suboptimal searches ($p << 0.01$) with no significant difference between blind search and LCRS.

While there is a clear trade-off between the approaches, the data we have presented so far is for one specific type of graph. In order to validate that these trends hold more generally, we also ran the same experiments for randomly generated STNUs.

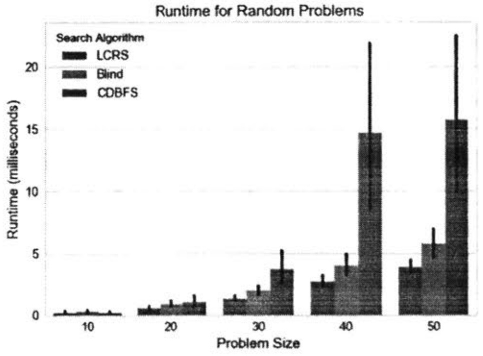


Figure 5-5: The runtimes of the solutions outputted by the search algorithms when run on random graphs.

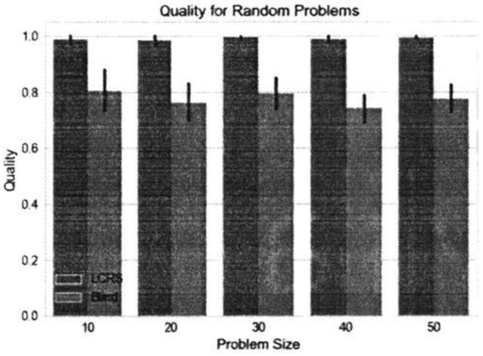


Figure 5-6: The quality of the solutions outputted by the search algorithms when run on random graphs. Quality is given by the optimal cost divided by the cost of the returned solution with a score of 1.0 representing the optimal solution.

Our experiments on random STNUs demonstrate that the differences we saw in runtime between the optimal and suboptimal approaches persist (Figure 5-5). At problem sizes of $k = 30, 40, 50$, CDBFS is slower than both suboptimal searches ($p < 0.05$). But in the random case, we also start to see a difference between blind search and LCRS. At $k = 30, 40, 50$, LCRS is significantly faster than blind search ($p << 0.01$).

When we turn our attention to cost, however, we see that on random graphs, we see a strong improvement in our results (Figure 5-6), as compared to the performance of our algorithms on the k-adversarial problems. Across all problem sizes, we see that on average blind search is within 35% of optimal, and even more remarkably, on average LCRS is within 1.5% of optimal on average. Given the massive difference in speed and the close approximation of optimal results, this provides strong support for the use of LCRS in low-cost communication window generation.

5.6 Conclusion

In this chapter, we introduced a series of techniques for generating solutions for CCMPs in both the offline and online instances. To produce communication windows that guaranteed successful execution, we provided an efficient means of extracting delay controllability conflicts and used those conflicts as a means of guiding our search through a continuous state space. The three algorithms we presented for the master problem in solving offline CCMPs, blind search, LCRS, and CDBFS, all have very different properties with the first two being significantly faster with the last one guaranteeing optimality. While we provide theoretical results demonstrating that the suboptimal searches can behave polynomially worse than CDBFS, in practice we expect that LCRS provides highly competitive results with significant improvements in speed.

Chapter 6

Chance-Constrained Variable Delays

In Chapter 3, we introduced formalisms for modeling uncertain delays in the observations of events through variable-delay controllability and chance-constrained variable-delay controllability. These models build heavily on delay controllability checking in STNUs (Chapter 4), and for each contingent event, we provide either a bounded interval or probability distribution representing the amount of time that may pass after an event occurs before an agent learns that it occurs. In this chapter, we provide a series of algorithms that solve these problems while offering three main contributions.

First, we provide an efficient algorithm for checking variable-delay controllability of STNUs. We approach the problem by creating a parallel STNU that is delay controllable with respect to a static delay function if and only if our original network is controllable. If we let n be the number of events in our schedule and m be the number of constraints, we can apply this transformation in $O(m+n)$ time, and since controllability checking under fixed observation can be performed in $O(n^3)$ time [3], we can similarly check variable-delay controllability in $O(n^3)$ time. Along with a controllability check, we also show how to derive an execution strategy for the online scheduling of controllable networks with uncertain observations. The same approach used to assess the controllability of the network yields a new, less expressive network that is controllable with respect to some fixed-delay observations. We show how only a few modifications are needed to reduce the variable-delay execution problem to that of finding an execution strategy on the less expressive network.

Second, we augment variable-delay controllability by adding risk bounds and by considering chance-constrained variable-delay controllability of temporal problems. Variable-delay controllability makes use of set-bounded notions of uncertainty and discards important distributional data that can help make quantitative probabilistic claims about the chance of success during execution. We show how to efficiently reason over variable distributions in the uncertain observation of events and provide algorithms for constructing such strategies.

Finally, we provide an empirical characterization of the quality of variable-delay controllability as contrasted against controllability checks that approximate the model using fixed delays. We show that even for the best approximations, the false positive rate is low but not zero, indicating that it is most appropriate to check variable-delay controllability directly. We supplement this work with an evaluation of chance-constrained variable controllability

The ability to model, validate, and execute temporal networks under observational uncertainty represents a unique and significant improvement over the state of the art. With previous temporal controllability formalisms, inference only flowed forward with time. Observations that happened in the future had no impact on our beliefs about past events. In contrast, our approach provides a rigorous means of incorporating future observations in our updated beliefs about past events.

6.1 Determining Controllability

We first start by considering the set-bounded variable-delay controllability problem. Our strategy for determining whether a given STNU S is variable-delay controllable with respect to a set-bounded $\bar{\gamma}$ is to instead construct a related STNU S' that is delay controllable with respect to a derived delay function γ'.

It is important to note that we do not expect S' and γ' to be expressive enough to model our original problem. Because our focus is on checking the controllability of our problem, it suffices for us to focus on the worst-case areas of our problem, or the situations that are hardest to schedule, and we construct S' and γ' with that end in mind. We prove that if it is possible to construct a policy that handles the worst-case areas, that same policy can be applied to other areas of the original STNU, not captured by S' and γ'.

To construct S' and γ', we start by copying the original graph S. We then make a series of iterative modifications to our new outputs so that they capture the controllability of our original input. We start by updating our definition of γ' based on observed values of $\bar{\gamma}$.

Lemma 6.1. *For any contingent event $e \in X_e$ in S, if $\bar{\gamma}^-(e) = \bar{\gamma}^+(e)$, we can express the same behavior in S' using $\gamma'(e) = \bar{\gamma}^+(e)$.*

Proof. If $\bar{\gamma}^-(e) = \bar{\gamma}^+(e)$, then $\bar{\gamma}$ already emulates a fixed-delay for event e. Assigning $\gamma'(e) = \bar{\gamma}^+(e)$ makes no change to the proposed controllability, since the modified S' and γ' cover the same scenarios as S and $\bar{\gamma}$. □

Lemma 6.2. *If $\bar{\gamma}^+(e) = \infty$ for some particular $e \in X_e$, we can express the same behavior in S' using $\gamma'(e) = \infty$*

Proof. If $\bar{\gamma}^+(e) = \infty$, then in some scenarios, e is unobservable in S. It does not matter that in some instances we may learn about e after some delay; controllability checking is about verifying that a valid execution strategy exists in all scenarios. Thus, if we can verify that S' is controllable when e is unobservable (when $\gamma'(e) = \infty$), then we know that S is controllable whenever e is observed after $t \in \bar{\gamma}(e)$, since a valid execution strategy could always choose to ignore the observation. If S' is uncontrollable when $\gamma'(e) = \infty$, then we also similarly know that we would not be able to find a valid execution strategy if e ended up unobservable in S. □

After repeated applications of Lemmas 6.1 and 6.2, we are left with a series of contingent events whose variable-delay function values represent finite ranges. Since the delay in observation is an uncertain and uncontrollable duration, a naive attempt at transforming the model would be to model observation itself as a contingent constraint (Figures 6-1a, 6-1b). This structure better equips us to reason about the controllability and execution of our original problem, but it is important to realize

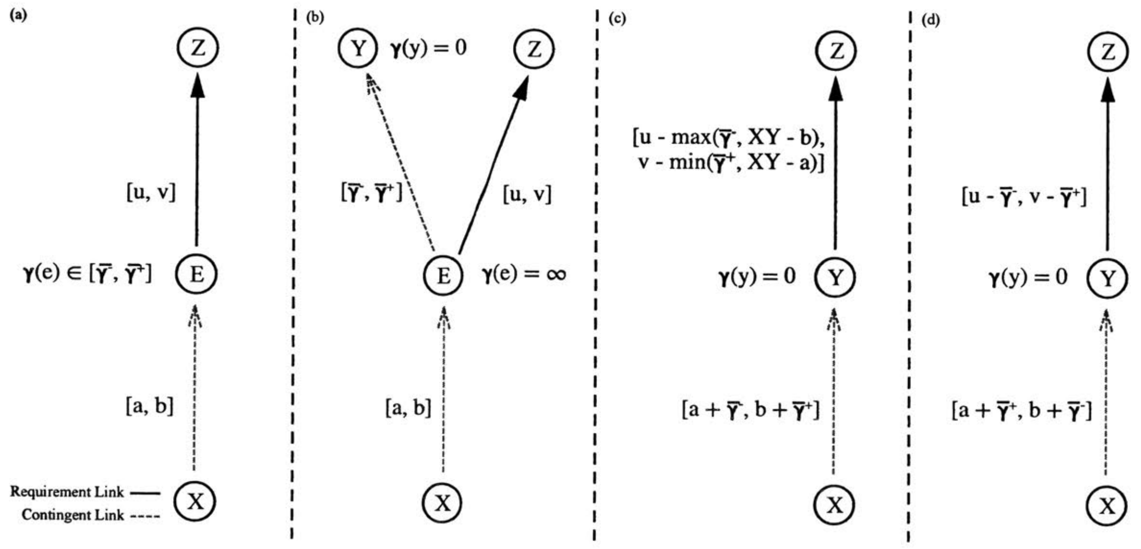


Figure 6-1: (a) A contingent constraint followed by a requirement constraint in our original STNU. (b) An equivalent (improper) STNU, which has a fixed-delay function instead of a variable-delay one. E becomes unobservable, and instead we immediately observe an explicit event Y after some uncertain delay. (c) An STNU that encodes a sufficient set of semantics to guarantee successful execution at runtime. XY refers to the true observed duration of the contingent constraint from X to Y. (d) A valid equivalent STNU, which has a fixed-delay function instead of a variable-delay one. The range of the contingent constraint shrinks, but the range of all attached requirement constraints must also shrink by a corresponding amount.

that this structure is not a valid STNU in the strict sense. In an STNU, all contingent constraints are required to start at an executable event, whereas this transformation lets a contingent constraint start at a contingent event. Nonetheless, while normal STNU algorithms may fail on this structure, we can still leverage it to better understand how to reason about variable-delay execution.

Under this new transformation, the next question we consider is how much information the observation of $\bar{\gamma}$ gives us in the worst case. Let $[a, b]$ be the bounds of the contingent constraint $X \Rightarrow E$. There are two ways to handle these contingent constraints based on how the uncertainty of the observation compares to the uncertainty of the original constraint. This depends on how the width of the bounds of the contingent constraint compares to the width of the bounds of possible delays in observation.

First we consider the case where the width of the bounds of possible delays in

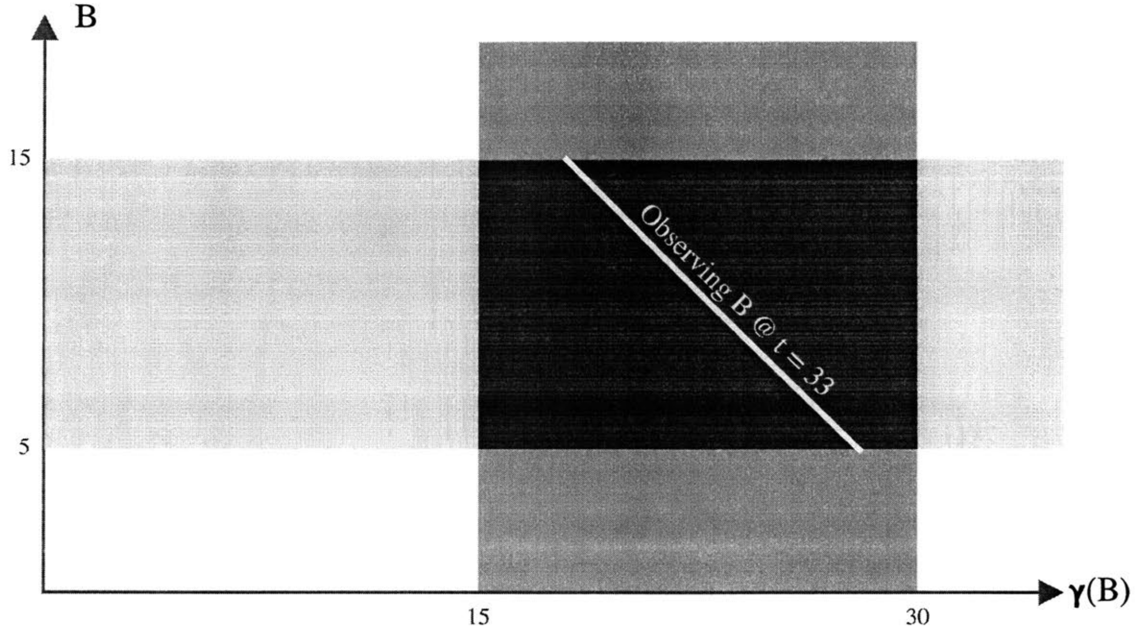


Figure 6-2: Here we consider a hypothetical execution of an STNU where contingent event B has $\bar{\gamma}(B) \in [15, 30]$, whereas the contingent constraint ending at B takes time in the range $[5, 15]$. There are some particular observations for which there is too much ambiguity to glean any information about the value of B.

observation is greater than the width of the bounds of the contingent constraint. We present a simple illustrative example in Figure 6-2.

In Figure 6-2, we consider a situation where the contingent link ending at B has duration in the range $[5, 15]$ while the delay in observation is in the range $[15, 30]$. If the event is observed at $t = 33$, then it is possible that B happened at the beginning of the range and the delay in observation was 28 minutes. But it could also be the case that the event occurred at at the latest possible time, but the delay in observation was 18 minutes. As a result, there are some observations that give information above the *a priori* knowledge of contingent link bounds. Since controllability is about guaranteeing success in all cases, if we can generate a dynamic schedule when there is no information, our strategy for doing so suffices, even if we can learn additional information. Thus, it suffices to solve for the controllability of a corresponding STNU where there is an infinite delay in observation for this particular event. We show how to prove and formalize this relationship in the following lemma.

Lemma 6.3. *If $b - a \leq \bar{\gamma}^+(e) - \bar{\gamma}^-(e)$, we can express the same behavior in S' using $\gamma'(e) = \infty$.*

Proof. In this situation, there is at least as much uncertainty in the observation of the event as there is in the occurrence of the event, meaning we have no guarantee of receiving any meaningful information after observing that the event happened. Consider the worst-case scenario where we learn that e happened $a + \bar{\gamma}^+(e)$ after the starting executable event. It is clear that the original contingent constraint could have taken on a value of a given this information, but because $a + \bar{\gamma}^+(e) - b \geq \bar{\gamma}^-(e)$, we also know that the original constraint could have had a value of b and thus been anywhere in $[a, b]$. Since $\bar{\gamma}$ is not guaranteed to give us any information in this instance, in our derived STNU S', we can let $\gamma'(e) = \infty$. □

We do not make controllability checking or network execution any harder by going from a highly uncertain observation to no observation at all. We always know the starting time of the original contingent constraint, as we choose it ourself, meaning we still have our original coarse bound $[a, b]$ on when the event occurs.

We next consider what happens if the width of the contingent link bound $[a, b]$ is narrower than the width of the bound on possible observation delay.

Lemma 6.4. *If $b - a \geq \bar{\gamma}^+(e) - \bar{\gamma}^-(e)$, we can replace the bounds of the original constraint ending at e with $[a + \bar{\gamma}^+(e), b + \bar{\gamma}^-(e)]$.*

Proof. If $b - a \geq \bar{\gamma}^+(e) - \bar{\gamma}^-(e)$, we can take our original contingent constraint with range $[a, b]$ and variable-delay $\bar{\gamma}(e)$, and in S' we can transform it into a contingent constraint $[a + \bar{\gamma}^+(e), b + \bar{\gamma}^-(e)]$ with $\gamma'(e) = 0$ (see contingent constraint across Figures 6-1a, 6-1b, 6-1d), folding some of the uncertainty in observing e directly into the contingent constraint.

It is important to notice that the range of the modified contingent constraint is shorter than the range of possible times at which we would actually notice the occurrence of e, $[a + \bar{\gamma}^-(e), b + \bar{\gamma}^+(e)]$. Here again, we rely on the notion that we are only interested in capturing the worst-case scenario. Imagine that we observed e at some time $a + \bar{\gamma}^+(e) - \epsilon$, where $0 < \epsilon \leq \bar{\gamma}^+(e)$. We know that the original constraint occurred at a lower-bound of a and that it has an upper-bound of $a+(\bar{\gamma}^+(e)-\bar{\gamma}^-(e))-\epsilon$. In contrast, with an observation at $a + \bar{\gamma}^+(e)$, we have a strictly larger range of possible options for the original contingent constraint, meaning our restriction does not make the scheduling problem less constrained.

We can make the same argument for the upper-bound. If we observe e at some $b + \bar{\gamma}^-(e) + \epsilon$, then we have an upper-bound of b and a lower-bound of $b - (\bar{\gamma}^+(e) - \bar{\gamma}^-(e))+\epsilon$. When we instead observe e at $b+\bar{\gamma}^-(e)$, we have the same upper-bound but a smaller lower-bound at $b - (\bar{\gamma}^+(e) - \bar{\gamma}^-(e))$, meaning the range of possible options is strictly larger. This means that our modified contingent constraint with $\gamma'(e) = 0$ fully captures the worst-case scenarios for e with our original S and $\bar{\gamma}$. □

What remains is to demonstrate how to transform the requirement constraints attached to e such that they represent the original execution semantics of S. To validate that this is the case, we first examine what local execution semantics look like in a variable-delay temporal network (Figures 6-1b, 6-1c).

Lemma 6.5. *If we have contingent constraint $X \Rightarrow E$ with duration $[a, b]$, outgoing requirement constraint $E \rightarrow Z$ with duration $[u, v]$ with an unobservable E, and contingent constraint $E \Rightarrow Y$ with range $[\bar{\gamma}^-, \bar{\gamma}^+]$, we can replace the original requirement constraint during execution with a new constraint $Y \rightarrow Z$ with bounds $[u - max(\bar{\gamma}^-, XY - b), v - min(\bar{\gamma}^+, XY - a)]$, where XY is the true duration of $X \Rightarrow Y$. See Figure 6-1c for reference.*

Proof. From an execution perspective, X and Y are the only events that can give us any information that we can use to reason about when to execute Z (since E is wholly unobservable).

If we execute Z based on what we learn from Y, then we use our information from Y to make inferences about the true durations of $X \Rightarrow E$ and $E \Rightarrow Y$, based on $X \Rightarrow Y$. We know that the lower-bound of $E \Rightarrow Y$ is at least $XY - b$ and that its upper-bound is at most $XY - a$. But we also have the a priori bounds on the contingent constraint that limit its range to $[\bar{\gamma}^-, \bar{\gamma}^+]$. Taken together, during execution we can infer that the true bounds of $E \Rightarrow Y$ are $[max(\bar{\gamma}^-, XY - b), min(\bar{\gamma}^+, XY - a)]$. Since we have bounds only on Z's execution in relation to E, we can then infer a requirement constraint $Y \rightarrow Z$ with bounds $[u - max(\bar{\gamma}^-, XY - b), v - min(\bar{\gamma}^-, XY - a)]$.

If we try to execute Z based on information we have from X, we must be robust to any possible value assigned to $X \Rightarrow E$. This means that we would be forced to draw a requirement constraint $X \rightarrow Z$ with bounds $[u + b, v + a]$. But we know that $u - max(\bar{\gamma}^-, XY - b) \leq u + b - XY$ and $v - min(\bar{\gamma}^-, XY - a) \geq v + a - XY$, which means that the bounds we derived from Y are at least as expressive as the bounds that we would derive from X. □

Since we have a local execution strategy that depends on the real value of XY, we can try to apply that strategy to the contingent constraint we restricted in Lemma 6.4, in order to repair the remaining requirement constraints.

Lemma 6.6. *If we have an outgoing requirement constraint $E \rightarrow Z$ with duration $[u, v]$, where E is a contingent event, we can replace the bounds of the original requirement constraint with $[u - \bar{\gamma}^-, v - \bar{\gamma}^+]$. See Figure 6-1d for reference.*

Proof. If we directly apply the transformation from Figure 6-1c to our original STNU, we introduce a new complexity in the form of reasoning over *min* and *max* operations in our constraint bounds. However, from Lemma 6.4, we know that in a controllability evaluation context, it is acceptable for us to have the $X \Rightarrow Y$ constraint take on stricter range $[a + \bar{\gamma}^+, b + \bar{\gamma}^-]$, instead of $[a + \bar{\gamma}^-, b + \bar{\gamma}^+]$, meaning for the purposes of evaluating controllability, we can assume $a + \bar{\gamma}^+ \leq XY \leq b + \bar{\gamma}^-$. When we evaluate the requirement constraint $Y \rightarrow Z$, we find that $max(\bar{\gamma}^-, XY - b) = \bar{\gamma}^-$ and $min(\bar{\gamma}^+, XY - a) = \bar{\gamma}^+$. This gives us the bounds for the requirement constraint that we see in Figure 6-1d. □

Lemma 6.6 handles outgoing requirement edges connected to contingent events, but we also must handle incoming edges.

Corollary 6.6.1. *If we have an incoming requirement constraint $Z \rightarrow E$ with duration $[u, v]$ where E is a contingent event, we can replace the bounds of the original requirement constraint with $[u + \bar{\gamma}^+, v + \bar{\gamma}^-]$.*

```
Input: STNU $S$; variable-delay function $\bar{\gamma}$
Output: An STNU $S'$ and fixed-delay function $\gamma'$
Initialization:
1  $S' \leftarrow S.copy()$;
2  $\gamma' \leftarrow \{\}$;
ConvertToFixedDelay:
3  for $l \in S'.contingentConstraints()$ do
4      $e \leftarrow l.endpoint()$;
5      $a, b \leftarrow l.bounds()$;
6      if $\bar{\gamma}^{+}(e) == \infty$ or $\bar{\gamma}^{+}(e) == \bar{\gamma}^{-}(e)$ then
7          $\gamma'(e) \leftarrow \bar{\gamma}^{+}(e)$;
8      else if $b - a < \bar{\gamma}^{+}(e) - \bar{\gamma}^{-}(e)$ then
9          $\gamma'(e) \leftarrow \infty$;
10     else
11         $l.setBounds(a + \bar{\gamma}^{+}(e), b + \bar{\gamma}^{-}(e))$;
12         $\gamma'(e) \leftarrow 0$;
13         for $l' \in e.outgoingReqConstraints()$ do
14             $u, v \leftarrow l'.bounds()$;
15             $l'.setBounds(u - \bar{\gamma}^{-}(e), v - \bar{\gamma}^{+}(e))$;
16         for $l' \in e.incomingReqConstraints()$ do
17             $u, v \leftarrow l'.bounds()$;
18             $l'.setBounds(u + \bar{\gamma}^{+}(e), v + \bar{\gamma}^{-}(e))$;
19 return $S', \gamma'$
```

Algorithm 10: Algorithm for converting a variable-delay controllability problem to a fixed-delay controllability one.

Proof. A requirement constraint $Z \rightarrow E$ with bounds $[u, v]$ can be immediately rewritten as its reverse $E \rightarrow Z$ with bounds $[-v, -u]$. After reversing the edge, we can apply Lemma 6.6 to get $Y \rightarrow Z$ with bounds $[-v - \bar{\gamma}^{-}, -u - \bar{\gamma}^{+}]$, which we can reverse again to get $Z \rightarrow Y$ with bounds $[u + \bar{\gamma}^{+}, v + \bar{\gamma}^{-}]$. □

Now we put it all together and introduce Algorithm 10 as a complete algorithm for transforming an STNU S and variable-delay function $\bar{\gamma}$ into a corresponding STNU S' and fixed-delay function γ', where S is variable-delay controllable if and only if S' is fixed-delay controllable.

Theorem 6.7. *We can convert a variable-delay controllability problem into a corresponding fixed-delay controllability problem in $O(m + n)$.*

Proof. Initially in Algorithm 10 (lines 6-7), we handle Lemmas 6.1 and 6.2. Next, we check the condition set forth by Lemma 6.3 (lines 8-9) to see if in the worst-case the observation would fail to give us new information.

Finally, in the remaining part of the algorithm (lines 10-18), we transform all other contingent constraints and the requirement constraints that are associated with them. At line 11, we apply the transformation as specified by Lemma 6.4. At lines 13-18, we ensure that all outgoing and incoming requirement constraints are updated as specified in Lemma 6.6 and Corollary 6.6.1.

Overall, the transformation is efficient, running in linear time. Let n be the total number of events and let m be the total number of constraints in an STNU. There are at most $O(n)$ contingent constraints, meaning the operations spanning lines 4-12 are applied at most $O(n)$ times. For each requirement constraint in S', we modify its bounds at most once as an outgoing constraint (lines 13-15) and once as an incoming constraint (lines 16-18). This means that lines 13-18 are executed at most $O(m)$ times, meaning the total runtime is $O(m + n)$. □

Since we can convert any variable-delay controllability problem into an equivalent-valued fixed-delay controllability problem, we can use fixed-delay controllability checkers to assess the variable-delay controllability of an STNU after transformation.

Theorem 6.8. *Variable-delay controllability can be evaluated in $O(n^3)$ time.*

Proof. Since the transformation takes $O(m+n)$ time without changing the size of the output STNU and fixed-delay controllability checking takes $O(n^3)$ [3], the result is an $O(n^3)$ way to check variable-delay controllability for any STNU since $m \leq n^2$. □

6.2 Variable-Delay Execution

Our algorithmic transformation gives us an efficient way to determine whether an STNU is variable-delay controllable but the resulting transformation does not give us an execution strategy for our STNU.

Intuitively, we want to use the resulting STNU from our fixed-delay transformation to guide execution but we face some limitations. To simplify our controllability checking, we restricted the ranges of many of our contingent constraints. If we tried to execute the resulting STNU, we would likely encounter a violation, since it is possible for the world to violate the invariants of our STNU – namely that nature will respect the bounds of all contingent constraints. However, this problem is not insurmountable. We can use our transformed STNU as a guide for execution but amend our execution semantics slightly, to account for these discrepancies.

Theorem 6.9. *Deriving an execution strategy for a variable-delay controllability problem reduces to finding an execution strategy for a fixed-delay controllability problem.*

Proof. The problem lies in a few contingent constraints that in the transformed STNU have bounds $[a+\bar{\gamma}^+, b+\bar{\gamma}^-]$ but in the original STNU have implicit bounds of $[a+\bar{\gamma}^-, b+\bar{\gamma}^+]$. The reason we were allowed to restrict these bounds is that the execution of other actions in the STNU were not dependent on the endpoint of this longer contingent constraint, but rather on some other contingent constraint that had bounds $[a, b]$. In the restricted case, we always had less information about the true duration of the original contingent constraint, despite the fact that the range itself was smaller.

Armed with this knowledge, the remedy for our execution strategy is relatively straightforward. We follow the normal fixed-delay execution strategy for our derived STNU but with two exceptions. First, if the true duration of a contingent constraint is less than $a+\bar{\gamma}^+$, we buffer the response and act as if the duration was actually $a+\bar{\gamma}^+$. It is clear that waiting gives us no extra information.

Second, if more than $b+\bar{\gamma}^-$ time has passed and the contingent constraint has not reached completion, we act as if it actually reached completion at $b+\bar{\gamma}^-$. We can safely assume the earlier completion time because of the information it gives us. When $b+\bar{\gamma}^-$ time passes, the value of the original contingent constraint is somewhere in $[b-(\bar{\gamma}^+-\bar{\gamma}^-), b]$. If we were to learn about the value at a later moment, $b+\bar{\gamma}^-+\epsilon$, then the value of the original contingent constraint would be in $[b+\epsilon-(\bar{\gamma}^+-\bar{\gamma}^-), b]$, which is strictly tighter. Thus, if we assume an earlier completion time, we give ourselves a strictly harder problem, but we know it is still controllable because this still maps to our corresponding fixed-delay controllability problem.

These changes restrict the information we can learn about the original $[a, b]$ constraint, but since the system is still controllable with this restriction, our execution strategy remains valid. Thus, finding an execution strategy for an STNU with variable-delay function reduces to finding an execution strategy for an STNU with a fixed-delay function. □

6.3 Checking Chance-Constrained Controllability

Our work thus far has focused on how to determine whether it is possible to construct a schedule for a temporal network when the uncertainty over when events are observed is bounded. While this approach provides us a guarantee of robustness, it often overweights the tail-distribution events that are possible but severely unlikely. Instead, we often care to ask a related question, which is whether it is possible to construct a schedule for a temporal network that succeeds most of the time but incurs a risk of failure in some small set of instances [25, 46, 59].

In general, this problem is remarkably difficult, as it relies on reasoning about the joint probability of communication events during execution. Without any *a priori* conditions on the types of probability distributions made available, generalizing such reasoning is a hard problem. Instead, however, it is useful to consider how we can use our set-bounded variable-delay controllability tactics to determine the existence of a chance-constrained variable-delay controllability solution. If a temporal network is variable-delay controllable under a set-bounded interpretation, then we know it is similarly chance-constrained variable-delay controllable under the same parameters. If we then use our knowledge of the probability distributions to reduce the width of our set-bounded observation windows, without exceeding the risk bounds and while guaranteeing set-bounded controllability, then we have provided a guarantee of the network's chance-constrained variable-delay controllability. The rest of this section will elaborate this procedure in more depth.

It is worth noting that our approach for determining chance-constrained controllability is sound but incomplete. Every solution we find is guaranteed to be a correct one, but because our solution does not deeply consider the nuances of the probability distributions themselves, it is possible that a solution exists that uses its intimate knowledge of the nature of the probability distribution to provide a much lower risk of failure. The study of more common distributions and their impact on risk of failure will be the subject of future work.

6.3.1 Approach

Our approach for solving the chance-constrained variable delay controllability problem in many ways resembles the approach for solving CCMPs that we introduced in Chapter 5. We decompose our problem in two, introducing a master problem and a sub-problem.

The master problem is responsible for picking ranges over the distribution of possible communication, such that those ranges respect the given chance constraint. It does so by using a non-linear program (NLP) solver, whose goal is to maximize the probability of a given set of ranges subject to constraints on those ranges. The goal of the sub-problem solver is to check whether the STNU is variable-delay controllable with respect to the set-bounded interpretation of those ranges.

As we did in Chapter 5, we use conflicts returned by the sub-problem solver to guide the master problem in its enumeration of possible distribution ranges. In this case, we use *variable-delay controllability conflicts*, which are used to explain why the STNU is not variable-delay controllable with respect to the input set-bounded communication ranges. We extract resolutions for these variable-delay controllability conflicts and frame those resolutions as linear constraints for use by the NLP solver in the master problem.

In the rest of this section, we explain how to augment our variable-delay controllability checking algorithm to return conflicts that can be used to inform our master problem. We then introduce a method for enumerating solutions that resolve all known conflicts that can be used to solve the master problem.

Walkthrough

Before we go into the details of our algorithms, we start by walking through an example to understand how the chance-constrained variable delay controllability algorithm operates. The entry point to the procedure, as well as the operation of the master problem, is found in Algorithm 13.

The algorithms for the sub-problem are composed of two parts. In Algorithm 11, we show how to convert a variable-delay controllability problem to an ordinary delay controllability problem; this algorithm differs from Algorithm 10 in that it rewrites the input STNU using $\delta_{\bar{\gamma}}$, which represents the width of the range, and $\bar{\gamma}^+$, which represents the upper-bound of the range, instead of using $\bar{\gamma}^-$ and $\bar{\gamma}^+$. More detail on why this change of parameter is used is discussed in the next subsection, but note that these parameters are going to be the variables over which the NLP solver makes its decision. In Algorithm 12, we show how to transform a variable-delay controllability conflict into a set of possible resolutions, represented by linear constraints.

To illustrate the operation of these algorithms, we considered a simplified version of Example 3.4 (see Figure 6-3). In this problem, we have two agents, Alex and Sam. Sam walks into the kitchen to make coffee at 8am, and it takes between 15 and 30 minutes for the coffee to be ready. Alex is in another room, but wants to get coffee between 20 and 30 minutes after it is ready in order to have it at the perfect temperature.

For the purposes of this problem, we assume that Sam is guaranteed to send an email to Alex somewhere uniformly in the range of $[0, 15] \cup [25, 100]$ minutes after the coffee is ready. We walk through a sample algorithm trace to illustrate how we are able to arrive at the solution that communication over the range $[5, 15]$ guarantees success and maximizes probability mass.

Algorithm 13 starts by checking to see if it is possible to guarantee controllability over the entire communication range (line 2). It starts by converting the STNU S with variable-delay function $\bar{\gamma}$ into a corresponding STNU S' with delay function γ' using Algorithm 11 (Algorithm 13, line 5). It then checks to see if S' is delay controllable

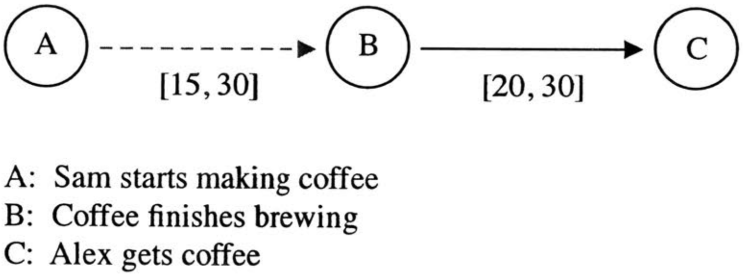
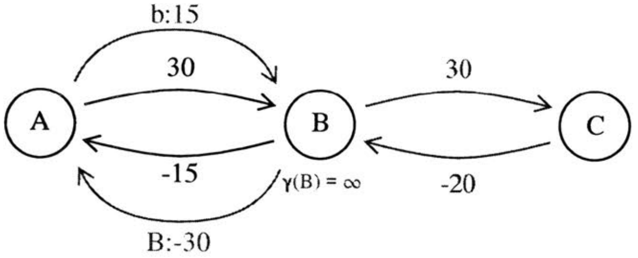


Figure 6-3: Simplified version of Example 3.4. The time it takes Sam to send an email is unspecified.



Current Values:

$\delta_\gamma(B) = \infty$, $\gamma^+(B) = \infty$

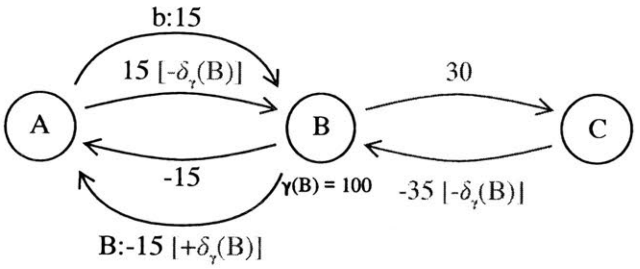
Known Conflict Resolutions:

- ~~[15 + 30 − 20 − 30 ≥ 0]~~ **or** [$\delta_\gamma(B) \leq 15$ **and** $\gamma^+(B) \neq \infty$]

Figure 6-4: The first step of the walkthrough. The algorithm checks to see if the problem is solvable with no communication. A conflict is found in red and the resolutions are noted.

with respect to γ' (Algorithm 13, line 6). In Figure 6-4, we see that the transformed STNU is not controllable, and the conflict that is extracted is highlighted in red.

Given a conflict, we must provide a series of resolutions for it so the master problem can generate better candidate ranges. In this case, Algorithm 12 gives us two possible resolutions that involve altering parameters $\delta_{\bar{\gamma}}$, which again represents the width of the window, and $\bar{\gamma}^+$, which represents the upper-bound of the range. The algorithm can either change the parameters to make the negative cycle positive (Algorithm 12, line 2), or it can change the parameters to ensure that $\gamma'(B) \neq \infty$ after transformation (Algorithm 12, line 5). The first option will never work, as our choice of parameters does not affect the weight of this particular cycle; note that this is implicitly handled by lines 12 and 13 of Algorithm 13, as the NLP solver will reject any program with this constraint as infeasible. Thus, we always must ensure that $\delta_{\bar{\gamma}}(B) \leq 15$ and $\gamma^+(B) \neq \infty$.



Current Values:

$\delta_\gamma(B) = 15$, $\gamma^+(B) = 100$

Known Conflict Resolutions:

- ~~[15 + 30 - 20 - 30 ≥ 0]~~ **or** [$\delta_\gamma(B) \leq 15$ **and** $\gamma^+(B) \neq \infty$]
- [30 - 20 - $\delta_\gamma(B) \geq 0$]

Figure 6-5: The second step of the walkthrough. The algorithm checks to see if the problem is solvable if communication is guaranteed to happen in the range [85, 100]. The conflict that is found involves an annotation, and its resolution is noted.

The NLP solver of our master problem can thus pick any window of size 15 from the domain $[0, 15] \cup [25, 100]$. We assume here that it picks [85, 100], setting $\delta_{\bar{\gamma}}(B) = 15$ and $\gamma^+(B) = 100$. When we apply the transformation to produce a new STNU and delay function, using Algorithm 11, we find another conflict (see Figure 6-5). In this case, the negative cycle is generated because of the choices of our parameter $\delta_{\bar{\gamma}}(B)$. In order to prevent this negative cycle from forming, we add the constraint $30 - 20 - \delta_{\bar{\gamma}}(B) \geq 0$ (Algorithm 12, line 2). Note that this is the first place that we used the annotations (see annotations in blue in Figure 6-5) that we added to our STNU (Algorithm 11, lines 12, 13, 18, & 21). The algorithm uses them to understand exactly how our choice of parameters affect the weights of different edges.

Given the new conflict resolution, our NLP solver must now also enforce that $\delta_{\bar{\gamma}}(B) \leq 10$, and we assume it picks the new range [90, 100]. We again find that the generated STNU is uncontrollable (see Figure 6-6). The two possible resolutions to the conflict that we extract require that the weights of the negative cycle become non-negative (Algorithm 12, line 2) or that the lower-case reduction involving edges $A \xrightarrow{b:15} B$ and $B \xrightarrow{30} C$ is eliminated (Algorithm 12, lines 7-12). Note that trying to make the cycle non-negative will again not work, as was the case in the first pass of the algorithm (Figure 6-4). Even though the weights of the edges are now different, the weight of the entire cycle has a $+\delta_{\bar{\gamma}}(B)$ term and a $-\delta_{\bar{\gamma}}(B)$ term from the edge annotations, which cancel each other out. The only way to resolve the conflict is to eliminate the lower-case reduction, which can be done by setting $\gamma^+(B) \leq 30$.

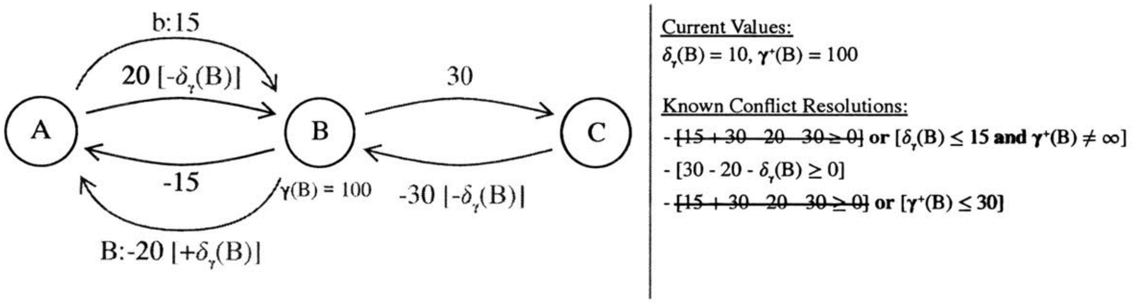

b:15
20 [-δ_γ(B)]
30
A
B
C
-15
γ(B) = 100
-30 [-δ_γ(B)]
B:-20 [+δ_γ(B)]
Current Values:
δ_γ(B) = 10, γ⁺(B) = 100
Known Conflict Resolutions:
- ~~[15 + 30 - 20 - 30 ≥ 0]~~ or [δ_γ(B) ≤ 15 and γ⁺(B) ≠ ∞]
- [30 - 20 - δ_γ(B) ≥ 0]
- ~~[15 + 30 - 20 - 30 ≥ 0]~~ or [γ⁺(B) ≤ 30]

Figure 6-6: The third step of the walkthrough. The algorithm checks to see if the problem is solvable if communication is guaranteed to happen in the range [90, 100]. The conflict stems from the fact that $\gamma^+(B) = 100$, and its resolution, that $\gamma^+(B) \leq 30$ is noted.

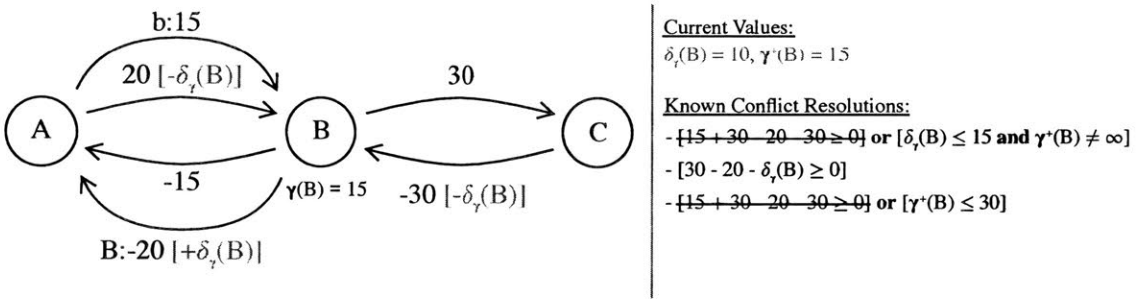

b:15
20 [-δ_γ(B)]
30
A
B
C
-15
γ(B) = 15
-30 [-δ_γ(B)]
B:-20 [+δ_γ(B)]
Current Values:
δ_γ(B) = 10, γ⁺(B) = 15
Known Conflict Resolutions:
- ~~[15 + 30 - 20 - 30 ≥ 0]~~ or [δ_γ(B) ≤ 15 and γ⁺(B) ≠ ∞]
- [30 - 20 - δ_γ(B) ≥ 0]
- ~~[15 + 30 - 20 - 30 ≥ 0]~~ or [γ⁺(B) ≤ 30]

Figure 6-7: The final step of the walkthrough. The conflict resolutions require that $\gamma^+(B) \leq 30$ and $\delta_{\bar{\gamma}}(B) \leq 10$. Our probability function assigns no probability mass to communication happening in the range [15, 25], so the algorithm checks whether the STNU is controllable when communication happens in the range [5, 15]. The STNU is controllable under these communication bounds, and the algorithm returns that as the solution.

As part of the final step, the NLP solver must pick a range for the distribution that satisfies $\delta_{\bar{\gamma}}(B) \leq 10$ and $\gamma^+(B) \leq 30$. While it could pick a range of [20, 30], our original probability function assumed that there was no probability mass in the range [15, 25]. As a result, the NLP solver will instead pick $[5, 15]$ as its range. When the sub-problem solver transforms the STNU and checks it for controllability, it finds that the resulting STNU is controllable, which means that if the range $[5, 15]$ is within our chance constraint, then the problem is chance-constrained variable-delay controllable.

We spend the rest of this section providing the proofs for these algorithms and their operation.

6.3.2 Finding and Resolving Conflicts

Even though our risk-bounding strategy is to take on risk by narrowing the uncertainty windows associated with the observation of each event, the problem of efficiently finding a satisfying set of windows is still difficult, as we are operating over a large set of continuous, interdependent variables. As such, we need a way to efficiently prune our search space to guide our search towards a more optimal result. Since our search strategy reduces to determining whether a choice of observation windows yields a set-bounded variable-delay controllable network, we turn our attention to deriving variable-delay controllability conflicts, which are responsible for explaining why our network is uncontrollable.

Our strategy for determining variable-delay controllability is to reduce the problem to an analogous network that is evaluated under fixed-delay controllability. In such an instance, our choice of values for observation windows has direct effects on both the duration of certain constraints (see Lemma 6.6) and the delay associated with those constraints (see Lemmas 6.1, 6.2, 6.3). When extracting conflicts associated with constraint durations, we can take inspiration from dynamic controllability conflict relaxation methods [62]. In particular, in order to relax the associated delays, we look to delay controllability relaxation methods [4].

We unfortunately cannot use these methods directly, as we need to tie our conflicts back to the underlying window choices that produced them. Further, we are only permitted to make a small subset of all possible modifications to resolve our conflicts. We know that in general the way to resolve variable-delay controllability conflicts is by reducing the width of the window or by sliding our window such that it occurs earlier, as increasing the width of the window and learning information later only makes it harder to construct an execution strategy. To make conflict extraction much more straightforward, for the purposes of conflict extraction, we represent our windows using $\bar{\gamma}^+$ and $\delta_{\bar{\gamma}}$, each of which is a function with domain X_e and range $\in \mathbb{R}^+ \cup \{\infty\}$, where $\bar{\gamma}^+$, as before, represents the upper-bound of the variable-delay window and $\delta_{\bar{\gamma}}$ represents the width of the variable-delay window. The constraint $\bar{\gamma}^+(x_c) \geq \delta_{\bar{\gamma}}(x_c)$ is

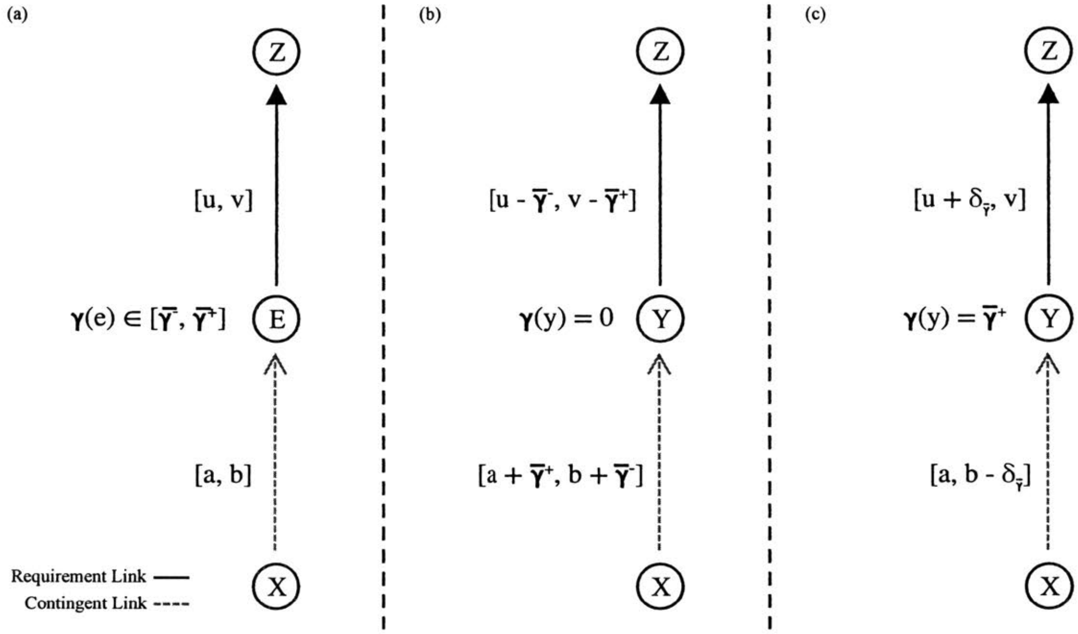


Figure 6-8: (a) A contingent constraint followed by a requirement constraint in our original STNU. (b) A valid equivalent STNU, which has a fixed-delay function instead of a variable-delay one. The range of the contingent constraint shrinks, but the range of all attached requirement constraints must also shrink by a corresponding amount. (c) Another equivalent fixed-delay STNU with its constraints instead parameterized in terms of $\bar{\gamma}^+$ and $\delta_{\bar{\gamma}}$.

required to ensure that there are no negative delays in observation. It is worth noting that $\delta_{\bar{\gamma}}$ can be equivalently defined by $\bar{\gamma}^+ - \bar{\gamma}^-$. Under this reparameterization, we can now consider adjustments to each variable independently rather than adjusting both to achieve the desired effect.

We now show how to transform our network using this alternative description before explaining the conflict extraction algorithm.

Lemma 6.10. *For any STNU S with variable delay function $\bar{\gamma}$, we can equivalently represent a contingent constraint $X \overset{[a,b]}{\Longrightarrow} E$ followed by a requirement constraint $E \xrightarrow{[u,v]} Z$ under variable-delay function $\bar{\gamma}$ (when $b-a > \bar{\gamma}^+(e) - \bar{\gamma}^-(e)$) by introducing a new event Y and using new edges, $X \overset{[a,b-\delta_{\bar{\gamma}}]}{\Longrightarrow} Y$ and $Y \xrightarrow{[u+\delta_{\bar{\gamma}},v]} Z$, with a new delay function γ' that enforces $\gamma'(y) = \bar{\gamma}^+(e)$. See Figure 6-8 for details.*

Proof. We perform this proof in two steps. First we show that the transformation of our original set of constraints into a pair $X \overset{[a+\bar{\gamma}^+,b+\bar{\gamma}^-]}{\Longrightarrow} Y$ and $Y \xrightarrow{[u-\bar{\gamma}^-,v-\bar{\gamma}^+]} Z$

with fixed-delay function $\gamma'(y) = 0$ (see transformation from Figure 6-8a to Figure 6-8b). Then we complete the transformation to the final edges, $X \xRightarrow{[a,b-\delta_{\bar{\gamma}}]} Y$ and $Y \xrightarrow{[u+\delta_{\bar{\gamma}},v]} Z$ with delay function γ', enforcing $\gamma'(y) = \bar{\gamma}^+(e)$. From Lemmas 6.4 and 6.6, we know that the transformation from Figure 6-8a to Figure 6-8b is sound. What remains is to show that the network in Figure 6-8b is executable if and only if the one in Figure 6-8c is as well.

First, observe what would happen if we moved event Y, which we refer to as a synthetic signaling event, by $\bar{\gamma}^+(e)$ units of time. We would have to shorten the lower and upper-bounds of $X \Rightarrow Y$ while simultaneously elongating the lower and upper-bounds of $X \rightarrow Z$ by $\bar{\gamma}^+(e)$. However, doing this changes the execution semantics, as the scheduling agent now has more time to plan to schedule Z. To offset this, we require $\gamma'(y) = \bar{\gamma}^+(e)$, so that Y is still observed by the scheduler at its original intended time. Under this interpretation, we are left with edges $X \xRightarrow{[a,b+\bar{\gamma}^- - \bar{\gamma}^+]} Y$ and $Y \xrightarrow{[u-\bar{\gamma}^- + \bar{\gamma}^+,v]} Z$. But of course, because $\delta_{\bar{\gamma}} = \bar{\gamma}^+ - \bar{\gamma}^-$, this is the same as $X \xRightarrow{[a,b-\delta_{\bar{\gamma}}]} Y$ and $Y \xrightarrow{[u+\delta_{\bar{\gamma}},v]} Z$. Thus, our transformation is equivalent under the new control variables $\bar{\gamma}^+$ and $\delta_{\bar{\gamma}}$.

□

We now introduce our modified algorithm, adapted from Algorithm 10, for converting a variable-delay controllability network to a fixed-delay controllability network (see Algorithm 11). This modified algorithm outputs a network that will be simpler for us to extract conflicts from when doing delay controllability checking. The main differences from our Algorithm 10 are lines 11, 14, 17, and 21; at these points, we perform the synthetic shifts in the observation of the event to allow us to parameterize our changes in terms of just $\bar{\gamma}^+$ and $\delta_{\bar{\gamma}}$.

Additionally, at this step we annotate the underlying labeled distance graph edges with values of $\delta_{\bar{\gamma}}$ that affect the edge's weights and annotate whether those values are used to increase or decrease the edge weight. We use these annotations when extracting conflicts to understand how best to resolve those conflicts. Specifically, for each edge in the distance graph that has an annotation, modifying the width of its stored $\delta_{\bar{\gamma}}$ will change the length of the edge, altering the available flexibility during

```
Input: STNU $S$; variable-delay function $\bar{\gamma}$
Output: An STNU $S'$ and fixed-delay function $\gamma'$
Initialization:
$S' \leftarrow S.copy()$;
$\gamma' \leftarrow \{\}$;
ConvertToFixedDelayWithWidth:
for $l \in S'.contingentConstraints()$ do
    $e \leftarrow l.endpoint()$;
    $a, b \leftarrow l.bounds()$;
    if $\bar{\gamma}^{+}(e) == \infty$ or $\delta_{\bar{\gamma}}(e) == 0$ then
        $\gamma'(e) \leftarrow \bar{\gamma}^{+}(e)$;
    else if $b - a < \delta_{\bar{\gamma}}(e)$ then
        $\gamma'(e) \leftarrow \infty$;
    else
        $l.setBounds(a, b - \delta_{\bar{\gamma}}(e))$;
        $l.upper.addAnnotation(-\delta_{\bar{\gamma}}(e))$;
        $l.labeledUpper.addAnnotation(+\delta_{\bar{\gamma}}(e))$;
        $\gamma'(e) \leftarrow \bar{\gamma}^{+}(e)$;
        for $l' \in e.outgoingReqConstraints()$ do
            $u, v \leftarrow l'.bounds()$;
            $l'.setBounds(u + \delta_{\bar{\gamma}}(e), v)$;
            $l'.lower.addAnnotation(-\delta_{\bar{\gamma}}(e))$;
        for $l' \in e.incomingReqConstraints()$ do
            $u, v \leftarrow l'.bounds()$;
            $l'.setBounds(u, v - \delta_{\bar{\gamma}}(e))$;
            $l'.upper.addAnnotation(-\delta_{\bar{\gamma}}(e))$;
return $S', \gamma'$
```

Algorithm 11: Algorithm for converting a variable-delay controllability problem to a fixed-delay controllability one.

execution.

Our method for extracting conflicts uses the same algorithms as in previous work for extracting delay controllability conflicts (see Chapter 5, Algorithms 3 and 4) . Our algorithms search for *semi-reducible negative cycles*, which indicate that a temporal network is uncontrollable and thus incapable of being executed. The set of edges associated with that cycle must then be modified, for example by a planner or relaxation algorithm, in order to produce a network that can be executed. In the case of determining chance-constrained variable-delay controllability, we cannot indiscriminately modify delays and temporal bounds in order to eliminate semi-reducible negative cycles. Instead, we must only modify those constraints whose values are affected by our choices of $\bar{\gamma}^+$ and $\delta_{\bar{\gamma}}$.

6.3.3 Finding Chance-Constrained Solutions

As described thus far, we apply existing conflict extraction algorithms for delay controllability problems and, while doing so, add annotations that map from edges of an STNU's labeled distance graph back to the variables that affect them. We now turn our attention to resolving those conflicts. When presented with a semi-reducible negative cycle, there are two possible ways to resolve the conflict. First, we can make the cycle a non-negative one, and second, we can eliminate the semi-reducibility of the cycle by changing an edge's values or the network's delay function so as to preclude a reduction from taking place. In this section, we introduce a conflict resolution algorithm (see Algorithm 12) and show how to apply these two tactics to generate conflict resolutions.

We start by examining our strategies for making a cycle non-negative. To eliminate the semi-reducible negative cycle by making it non-negative, we must find a way to adjust $\bar{\gamma}^+$ and $\delta_{\bar{\gamma}}$ such that $\sum\limits_{l} weight(l) \geq 0$, where l are the edges of the semi-reducible negative cycle. To do so, we can use the annotations directly, to rewrite our constraint as $\sum\limits_{l}\left(origWeight(l) + \sum\limits_{a \in l.annotations()} a\right) \geq 0$. In this instance, all of our annotations are of the form $\pm\delta_{\bar{\gamma}}(e)$ since $\bar{\gamma}^+$ has no impact on the length of edges. This resolution is added at line 2 of Algorithm 12.

We can take a similar approach to eliminate the semi-reducibility of a cycle. A cycle is semi-reducible if a series of reductions can be applied such that all lower-case edges are eliminated. In certain instances, it is possible to adjust the parameters of the STNU such that the returned cycle is still negative, but certain lower-case and cross-case reductions can no longer be applied. To identify such permutations, we add a new set of constraints that, if satisfied, eliminate the cycle.

To do so, we want to find the segment of the cycle that is responsible for eliminating a lower-case edge. For each lower-case edge l with label b, we march forward along the cycle, starting from l, until the total weight of the subpath is less than $\gamma'(b)$ and the subpath (including l) consists of at least two edges; this process is represented by the while loop at lines 9-11. This subpath generation is guaranteed to terminate, as the entire cycle has negative weight and for all e, $\gamma'(e) \geq 0$. We then add the constraint $\gamma'(b) \leq \sum_{l'} weight(l')$, where the summation iterates over all edges l' following the lower-case edge in our derived subpath. As before, we can represent the edge weight in terms of the original weight and the effects of $\bar{\gamma}^+$ and $\delta_{\bar{\gamma}}$ with $\gamma'(e) \leq \sum_{l'} \left(origWeight(l') + \sum_{a \in l'.annotations()} a \right)$.

While, as in the previous case, $\bar{\gamma}^+$ does not affect any of the edge weights, it does have an impact on the value of γ'. In the instance where $\gamma'(e) \neq \infty$, then we can simply write our constraint as $\bar{\gamma}^+(e) \leq \sum_{l'} \left(origWeight(l') + \sum_{a \in l'.annotations()} a \right)$ (line 13). However, if $\gamma'(e) = \infty$, we have to add new constraints based on the original values of $\bar{\gamma}^+(e)$ and $\delta_{\bar{\gamma}}(e)$. We can find ourselves in this situation if either $\bar{\gamma}^+(e) = \infty$ or if $\delta_{\bar{\gamma}}(e) > b - a$, so to resolve the conflict in this case, we require modifying our choices of variables jointly such that $\delta_{\bar{\gamma}}(e) \leq b - a$ and $\bar{\gamma}^+(e) \neq \infty$ (line 5). This allows us to potentially circumvent the lower-case and cross-case reductions by ensuring that $\gamma'(e) \neq \infty$ going forward.

It is important to note that the possible resolutions that we extract from our conflicts may neither be necessary nor sufficient to guarantee the executability of our network. To address this, we search across candidate permutations, using conflict-directed search, to find a permutation of $\bar{\gamma}^+$ and $\delta_{\bar{\gamma}}$ that satisfies our risk bounds. We can be further improve our search process. The cost of taking a resolution is the risk we incur by ignoring the tail ends of $\bar{\gamma}$'s distribution. By incorporating this cost in a variant of conflict-directed A* [60], we can significantly speed up our process.

Our final algorithm, used to solve the chance-constrained variable-delay controllability problem, is inspired by previous work in solving chance-constrained Probabilistic Simple Temporal Networks [59] and interleaves the use of a nonlinear program solver with conflict-directed search in order to determine whether a given STNU is controllable with respect to some risk bound (see Algorithm 13). Notably our

```
Input: A semi-reducible negative cycle, $C$;
existing observation window parameters $\bar{\gamma}^+$ and $\delta_{\bar{\gamma}}$;
current delay function $\gamma'$
Output: A set of possible conflict resolutions
Initialization:
1  $resolutions \leftarrow []$; list of conflict resolutions;
ExtractResolutions:
2  $resolutions.add\left(\sum_{l \in C}\left(origWeight(l) + \sum_{a \in l.annotations()} a\right) \geq 0\right)$;
3  for $l \in C.lowerEdges()$ do
4      if $\gamma'(l.label) == \infty$ then
5          $resolutions.add\left((\delta_{\bar{\gamma}}(l.label) \leq l.upper() - l.lower()) \wedge (\bar{\gamma}^+(l.label) \neq \infty)\right)$;
6      else
7          $next \leftarrow C.next(l)$;
8          $subpathEdges \leftarrow [l, next]$;
9          while $subpathEdges.weight() > \gamma'(l.label)$ do
10             $next \leftarrow C.next(next)$;
11             $subpathEdges.append(next)$;
12         $resolutions.add\left(\bar{\gamma}^+(l.label) \leq \sum_{l' \in subpathEdges}\left(origWeight(l') + \sum_{a \in l'.annotations()} a\right)\right)$;
13 return $resolutions.filter(r \rightarrow (r(\langle\bar{\gamma}^+, \delta_{\bar{\gamma}}\rangle) == false))$
```

Algorithm 12: Algorithm for extracting possible resolutions for a reported semi-reducible negative cycle.

constraints are all linear, and nonlinearity only comes into play when assessing the probability density function. When the function itself is smooth, it is straightforward to use off-the-shelf solvers, such as IPOPT [9] or SNOPT [27], in our implementation.

The algorithm works as follows. We initially consider the full range of values that the variable-delay function $\bar{\gamma}$ could take on (line 2). At each step in the search process, variable-delay controllability is checked by transforming it to a delay controllability problem (line 5) and using Algorithm 1 from Chapter 4 to check controllability (line 6). If the network is controllable, the algorithm returns immediately, reporting true if the solution is within our risk bound (line 8). The use of A* to guide the search optimally guarantees that we end at the lowest-cost approximation, and because our window approximation is conservative, we have a guarantee that our algorithm is

```
Input: STNU $S$, variable-delay function $\bar{\gamma}$ for which a probability distribution
       function $p$ is defined, tolerated level of risk $\Delta$;
Output: True if $S$ is chance-constrained variable-delay controllable with respect
        to risk tolerance $\Delta$, and false otherwise;
Initialization:
1  $queue \leftarrow []$ // priority queue of constraints to enforce on restrictions to $\gamma$;
ChanceConstrainedVariableDelayControllable?:
2  $queue.append(\langle \emptyset, \bar{\gamma} \rangle, 0)$;
3  while $queue.size() > 0$ do
4      $constraints, \bar{\gamma}', cost \leftarrow queue.popMin()$;
5      $S', \gamma' \leftarrow$ ConvertToFixedDelayWithWidth($S, \bar{\gamma}'$);
6      $controllable, conflict \leftarrow$ FixedDelayControllable($S', \gamma'$);
7      if $controllable$ then
8          return $true$;
9      for $r \in$ ExtractResolutions($conflict, \bar{\gamma}', \gamma'$) do
10         $newConstraints \leftarrow constraints.copy()$;
11         $newConstraints.append(r)$;
12         $\bar{\gamma}'', obj \leftarrow NLP.findMinimalWindows(newConstraints)$;
13         if $obj \neq \emptyset$ and $obj \leq \Delta$ then
14             $queue.append(\langle newConstraints, \bar{\gamma}'' \rangle, obj)$;
15 return $false$;
```

Algorithm 13: Algorithm that evaluates whether an STNU is chance-constrained variable-delay controllable with respect to risk bound Δ.

sound; whenever it returns true, there does exist an execution strategy within the given risk bounds.

In the event that the network is not controllable, a conflict is extracted and possible adjustments to $\bar{\gamma}^+$ and $\delta_{\bar{\gamma}}$ are enqueued in the form of a new $\bar{\gamma}$. However, rather than just enqueueing the values, we also enqueue the derived constraints. This approach is important because it allows us flexibility when determining the cost of our window tightening. To determine the cost of satisfying a new constraint, we frame a nonlinear program with one of the new constraints derived from the conflict as well as all of the constraints that were passed in as part of this state space (line 12). Given these constraints, we ask our nonlinear program solver to provide a solution subject to the objective of maximizing the overall probability of values falling within that window. Thus, our process is able to fluidly adjust its window according to the particularities of the probability distribution as well as the set of conflicts we have derived.

	Variable-delay controllable	Variable-delay uncontrollable
Min-fixed controllable	245	292
Min-fixed uncontrollable	0	463
Mean-fixed controllable	245	23
Mean-fixed uncontrollable	0	732
Max-fixed controllable	245	15
Max-fixed uncontrollable	0	740

Table 6.1: Variable-delay vs. minimum, mean, and maximum fixed-delay controllability and results when using an exponential delay function with $\lambda = 0.5$.

6.4 Empirical Evaluation

In this section, we provide empirical evaluations of our variable-delay controllability checking algorithms, showing first that variable-delay controllability gives us a level of modeling expressiveness that cannot be captured by approximations that use delay controllability alone. We second show that our chance-constrained variable-delay controllability algorithm empirically conforms to established risk bounds.

6.4.1 Controllability Experiments

The introduction of variable-delay controllability gives us a level of expressiveness that we previously lacked. In this section, we attempt to characterize the gap in expressiveness by showing how attempting to evaluate variable-delay problems using other models as approximations leads to incorrect results.

To evaluate the comparative quality of the different approaches, we construct a set of randomly generated STNUs. Each STNU has 10 contingent constraints with lower-bound 0 and an integer upper-bound uniformly chosen between 1 and 4. Each contingent constraint has a variable-delay function with a lower-bound of 0 and upper-bound chosen from the exponential distribution $f(t) = \lambda e^{-\lambda t}$ with $\lambda = 0.5$. For each pair of contingent constraint endpoints, we establish a requirement constraint between them with probability $\frac{1}{40}$. Each requirement constraint has a lower-bound of 0 and an integer upper-bound uniformly chosen between 1 and 4. We choose these parameters because they represent a reasonable trade-off between simplicity in degenerate cases and sufficient complexity to exhibit interesting behaviors.

We employ three different strategies for our fixed-delay approximations: $\gamma(x_c) = \bar{\gamma}^-(x_c)$, $\gamma(x_c) = \frac{\bar{\gamma}^- + \bar{\gamma}^+}{2}$, and $\gamma(x_c) = \bar{\gamma}^+(x_c)$. For each strategy, we know that whenever the original STNU is variable-delay controllable with respect to $\bar{\gamma}$, it is also fixed-delay controllable with respect to γ. Each choice of γ represents a potential realization of the delays offered by $\bar{\gamma}$, and the fixed-delay approximation has the added benefit of eliminating uncertainty in observation.

We generate 1000 different STNUs and compare the variable-delay controllability results to the different fixed-delay controllability approaches (Table 6.1). The instances that are of greatest interest are those where the STNU is not variable-delay controllable but the fixed-delay approximations determine it to be controllable.

This false positive rate of the minimum fixed-delay controllability approximation is quite high, at 39%. The mean and maximum fixed-delay approximations have more reasonable false positive rates at 4.5% and 3.0%, respectively. Since all approximations yield the correct answer when the original STNU is variable-delay controllable, it makes sense that the maximum fixed-delay approximation has the lowest false positive rate, as it is the most demanding of the three.

We note that these results are also dependent on the width of the variable-delay ranges found in the network. We can increase the likelihood that a delay takes longer by decreasing the choice of λ in our exponential delay function. When we vary our delay function using $\lambda = 0.5, 0.1, 0.05, and 0.01$, the false positives of the max-delay approximation are 3.0%, 3.4%, 3.9%, and 5.2%, respectively. As expected, this indicates that as the uncertainty in our delay grows, there is an increasing advantage, from a correctness perspective, to using variable-delay controllability.

In addition to simulating the network using fixed-delays, we also consider the effect of combining the two sources of uncertainty, the duration of the action and the delay in observation, into one new source of uncertainty. Unlike the fixed-delay

	Variable-delay controllable	Variable-delay uncontrollable
Elongated controllable	144	0
Elongated uncontrollable	101	755

Table 6.2: Variable-delay controllability vs. the controllability of a network that elongates its contingent constraints to account for observational uncertainty when using an exponential delay function with $\lambda = 0.5$.

approximations, we know that if a network under this transformation is controllable, then so too is the original network, as this approach discards any existing knowledge about the difference in uncertainties between the original event and the observation of that event.

As seen in Table 6.2, this approach yields no false positives, but unfortunately has a high false negative rate of 41.2%. An appropriate approximation strategy can be adopted to prevent either false positives or false negatives; however, such a wide disparity in results strongly reinforces the value of modeling observational uncertainty directly.

6.4.2 Chance-Constrained Experiments

In this section we evaluate our chance-constrained variable-delay controllability algorithms, with particular attention paid to the speed with which we are able to find a solution, as well as how close our derived execution scheme is with respect to the risk bound.

As a representative example, we elect to model the efficiency of a rental car agency in a problem setup very similar to the Zipcar problem, adapted to chance-constrained temporal networks [12, 25]. In this scenario, a rental car shop rents out each car for a specified number of trips per day and wants the cars out for as long as possible, in order to maximize revenue. Since the trip durations are largely determined by the routes picked by the renters, the only real control the renters have is in the time spent between when the car is dropped off and when it is picked up again. It takes a minimum of five minutes to clean and inspect the car, and management wants the car to sit on the lot for no more than 30 minutes, before it is given to the next customer.

Complicating this procedure is the fact that the attendants checking in the cars are different from the ones interacting with the customers. Depending on who is checking in the cars, the customer-facing worker at the front desk will learn that certain cars are checked back in at different times. Worker #1 reports that the car has been checked in immediately, 99% of the time, and forgets to report about the car's status, 1% of the time. Worker #2 is lazier and reports the status of the car after 15 minutes, 99% of the time, and similarly forgets to report the car's status, 1% of the time. Worker #3 generally reports the car's status after 15 minutes, doing so 90% of the time, but never forgets, reporting the car's return immediately, 5% of the time, and reporting it after 30 minutes, the remaining 5% of the time.

In our experiments, we craft scenarios where cars make one of 2, 5, 10, 15, 20, or 25 trips in total. For each choice of parameters, we created 100 distinct scenarios, and in each scenario, each car return is chosen to be processed randomly by a different worker. Since this scenario has no reliable guarantee that all constraints can be met, we set an upper-bound for risk at 15% and evaluate how close our algorithm is to the proposed risk bounds. All experiments were run on an Intel i7 processor with 4 cores and on a machine with 40GB of RAM.

The results of our experiments fall in line with our expectations of the system and demonstrate the utility of such a system in practice. As expected, we see the risk of failure grow in a linear fashion with the number of trips scheduled per car (see Figure 6-9). Despite the fact that our chance-constrained variable-delay controllability algorithm only allocates risk in single contiguous windows, it is still capable of finding reasonable solutions within the specified risk bound.

Importantly, we also examine the runtime of the algorithm to understand whether it is fast enough to be used in a practical sense (see Figure 6-10). The chance-constrained variant of the variable-delay controllability checker explicitly searches over different windows and as such is expected to have an exponential growth curve. We see this in our empirical results, as once we scale up to 25 trips per car, determining

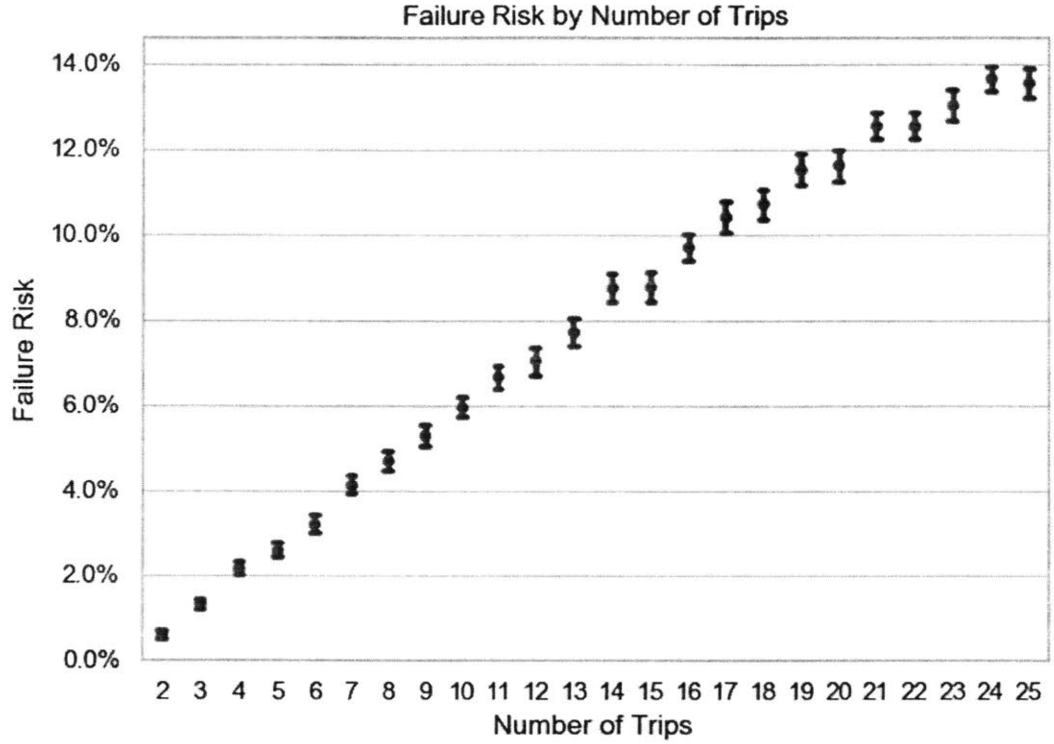


Figure 6-9: Empirical comparison of the risk of failure versus the number of trips considered in a single plan.

whether we can satisfy our risk bound takes nearly 20 seconds. Most activities have on the order of low tens of activities per agent and involve execution times that span hours; having a chance-constrained variable-delay controllability checker that runs in under a few seconds for typical networks, and in under a minute for even the most complex networks, implies that the algorithm is sufficiently fast for use in larger systems.

6.5 Discussion

In this chapter, we introduced variable-delay controllability as an extension to fixed-delay controllability over STNUs. We provide a formal definition showing how it generalizes fixed-delay controllability, while also providing an efficient sound and complete algorithm for determining the variable-delay controllability of an STNU.

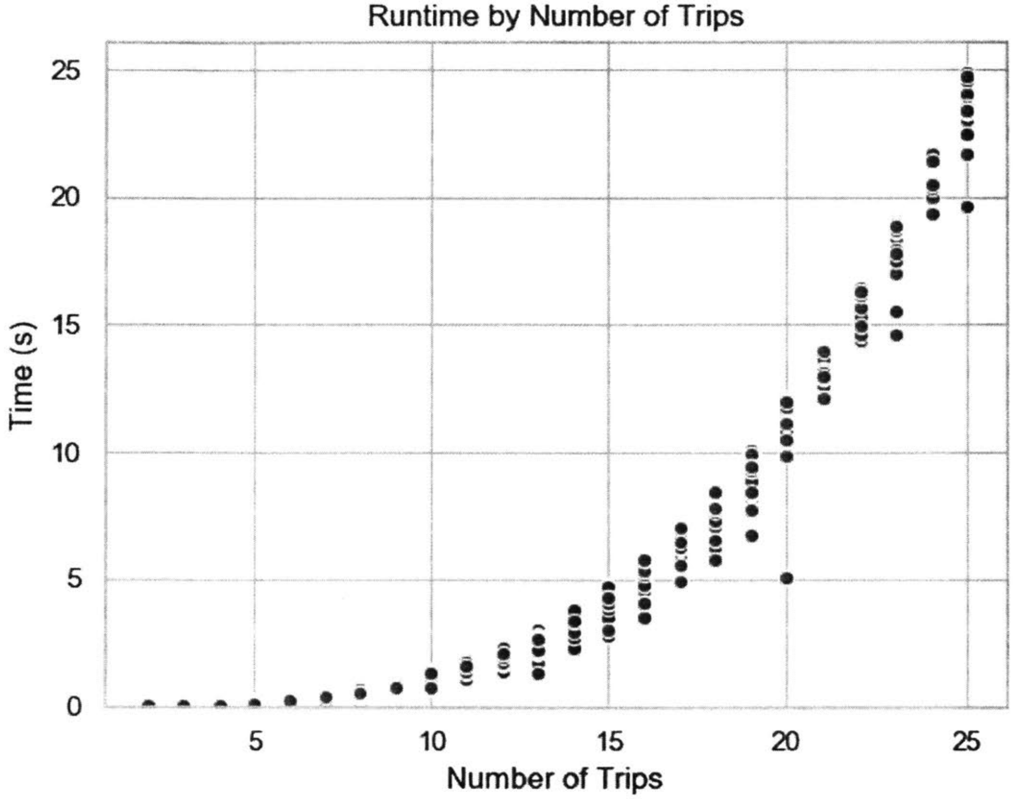


Figure 6-10: Empirical considerations of the runtime required to find a minimum-risk plan versus the number of trips included in a single plan.

Because variable-delay controllability execution reduces to fixed-delay controllability execution, we are able to demonstrate that variable-delay controllability is a formalism that can be used in practice to construct and evaluate schedules in the face of both temporal and observational uncertainty.

Chapter 7

Concluding Remarks

7.1 Contributions

In this thesis, we aimed to improve the function of real-time executives in multi-agent scenarios. Of particular importance to us was the handling of multi-agent scenarios where there was some sort of limit on communication that made it so communication was not guaranteed to happen instantaneously. The work presented in this thesis provides a set of definitions, algorithms, and corresponding theoretical analysis that demonstrates how to build a temporal executive capable of operating in a communication-limited world.

In Chapter 3, we laid out our desiderata and a framework satisfying it. At its core, we wanted a framework that could represent delays between actions and communication around those actions, describe the cost associated with the communication (and the corresponding problem to find low-cost communication windows), and be robust to imprecise communication. Respectively, we solved these problems by introducing delay controllability, the Communication Cost Minimization Problem, and variable-delay controllability. In subsequent chapters, we provided efficient algorithms for handling the types of problems that the framework satisfying our desiderata was mean to address.

In Chapter 4, we introduced a series of algorithms for checking delay controllability, which allowed us to incorporate notions of communication delay that are common in multi-agent scenarios to our temporal networks while still guaranteeing feasibility checking and execution in polynomial time.

Delay controllability gave us a means to determine whether we could construct a satisfying schedule given a planned communication strategy, but it did not provide us a straightforward way to construct such a strategy. In Chapter 5, we addressed this problem and showed how to adapt conflict-directed search methods to generate valid communication plans, subject to some cost function. Of importance, we showed that while certain suboptimal algorithms can give polynomially bad approximations of the optimal communication strategy in theory, in practice these algorithms are significantly faster and provide results that are near optimal. We then introduce a procedure for reactively adapting these communication strategies. When dealing with real agents, slight deviations from a plan need to be handled in order to ensure robust execution. Our work shows how to maintain a temporal horizon to preserve plan feasibility for as long as possible.

Finally, we built on the notion that our executive should be robust by adding in the capability to handle noisy communication. In Chapter 6, we introduced variable-delay controllability as a means of representing communication about events that have some uncertainty associated with it. We showed that this problem can be reduced to checking for ordinary delay controllability in the set-bounded case, and when given a distribution over communication noise, we considered the notion of checking chance-constrained variable-delay controllability. Our approach interleaved conflict-directed search with the use of a non-linear program solver and is fast enough for practical use.

7.2 Future Work

As we turn our attention forward, there are many interesting extensions and continuations of this work that are worth pursuing.

We expect that in the future significantly more work will be done on expanding the controllability checking problem to different types of communication models. While the formalism that we have presented with delay controllability is powerful, it makes relatively simple assumptions about the nature of communication. We assume that the success, failure, and delays in communication across all events are independent of one another and that they are simply a function of the original event. Including additional conditions on communication, like we did in Chapter 5 when describing flaky network outages, requires separate independent reasoning on top of the existing algorithms. We expect that as this model is tested and expanded, the need for richer communication models may be apparent. Communication may only be permissible in certain windows, may come as a large batched response, or may vary across time as specified by some distribution. Our work provides a strong foundation that should help future researchers as they choose to study richer communication models.

Another avenue for future research is to more directly tackle the problem of understanding the controllability of POSTNUs. Modelers who hope to use planners and executives to tackle problems select their models based on the trade-off between speed of evaluation and expressiveness of the model. While this thesis has put in a significant amount of work to prove theoretical bounds for many types of networks that modelers might choose, the POSTNU (and MaSTNU) remain open questions.

In many ways, however, the work of this thesis may provide insights as to whether checking POSTNU controllability is a problem that may be tractable. The work done on checking delay controllability and notably on checking variable-delay controllability represent significant advantages over the existing state of the art. Practical algorithms for checking the controllability of POSTNUs only exist if they are known to be chain-free [10], but our work on variable-delay controllability demonstrates how it might be possible to augment this controllability checking for certain types of chained contingent constraints. There is, of course, much more nuance to this problem, but we are optimistic that there is exciting work in this space ahead.

Appendix A

Controllability Complexity for Different Temporal Networks

In temporal planning, many different temporal formalisms are used to model real world situations and, in particular, can be used to varying different degrees to encode and reason over multi-agent execution. The choice of any particular type of network in modeling a problem has inherent trade-offs. If a temporal model supports more features, it can model a given scenario with higher fidelity. However, the additional features come at the expense of performance; modelers care about constructing schedules for temporal networks, and the presence of additional feature types can dramatically slow the runtime of scheduling algorithms.

Disjunctive constraints are important for modeling common phenomena like resource constraints and mutual exclusion (i.e. I can eat 30 minutes before swimming or after, but cannot eat while in the pool). These types of networks have been studied extensively in the forms of Temporal Constraint Satisfaction Problems (TCSPs) [22] and Disjunctive Temporal Networks (DTNs) [49]. Another important feature that is needed to faithfully model non-determinism in temporal events, such as the effect of traffic on a drive across town, is temporal uncertainty. Temporal uncertainty and disjunction have been studied together in Temporal Constraint Satisfaction Problems with Uncertainty (TCSPUs) [55] and Disjunctive Temporal Networks with Uncertainty (DTNUs) [57]. Finally, it is possible to directly model multi-agent interactions using Multi-agent Simple Temporal Networks (MaSTNs) [11], Multi-agent Simple Temporal Networks with Uncertainty (MaSTNUs) [15], and Partially Observable Simple Temporal Networks with Uncertainty (POSTNUs) [36]. The computational complexities of many of the simpler variants of these temporal models have been well-studied, but the same cannot be said of more advanced models. Despite this gap, there has been considerable effort put into constructing improved algorithms for these feature-rich temporal networks [16, 17, 18, 56].

In this appendix, we examine the theoretical complexity bounds of computing the controllability of many of these types of temporal networks. The first part of this appendix considers networks that feature conditional constraints, disjunctive constraints, and temporal uncertainty and reflects work originally published in AIJ [8]. The second part of this appendix considers networks with partial observability and temporally uncertain events and reflects work originally published in AAMAS [7]. The existing bounds for some of these results have been quite loose with most decision problems not known to be better than EXPTIME and some not known to be better than EXPSPACE. Our results are summarized in Figures A-1 and A-5 and represent a significant improvement over the best-known bounds. We finally conclude with a discussion of our results, giving practical advice to modelers who are interested in the trade-offs of using different temporal networks and lending insight into the differences between these networks.

There are many types of temporal networks beyond those that we focus on in this appendix. Many include features related to actor decisions, such as Temporal Plan Networks [34], Temporal Plan Networks with Uncertainty [33], Controllable Conditional Temporal Problems [63], Conditional Simple Temporal Networks with Decisions [13], and Conditional Simple Temporal Networks with Uncertainty and Decisions [64] while others, such as Probabilistic Simple Temporal Networks [25] and their relevant extensions, consider probabilistic temporal bounds. Despite the existence of other networks our work covers a broad area of focus that is under active investigation. Future work in this direction will focus on characterizing, organizing, and providing tighter bounds for controllability in these other types of networks but

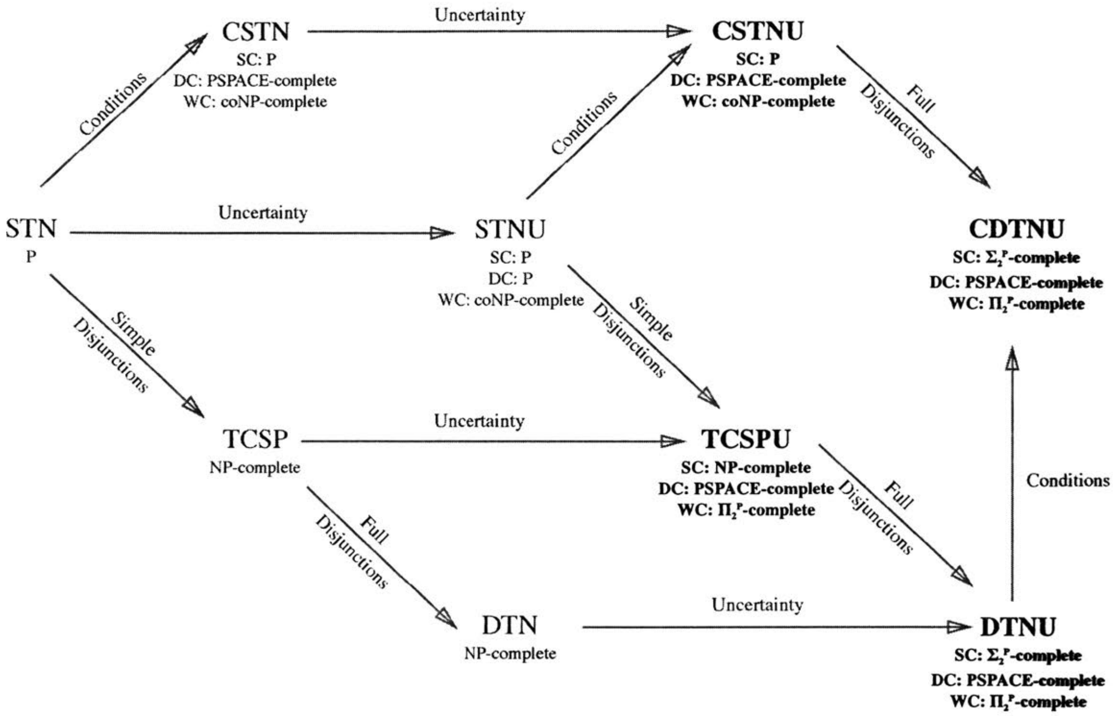


Figure A-1: A taxonomic organization of temporal networks considered in the first section of this appendix, how they relate to one another, and the complexity classes to which their decision problems belong. SC, DC, and WC represent strong controllability, dynamic controllability, and weak controllability, respectively. Results in bold represent novel results provided in this thesis.

is outside the scope of this thesis.

A.1 Conditional & Disjunctive Networks

In this section, we consider the theoretical complexity bounds of networks that feature conditional and disjunctive constraints as well as temporal uncertainty. The full set of networks and their corresponding set of results are summarized in Figure A-1. We divide the discussion of temporal networks into that of base temporal networks, which build on the simplest temporal network representations, and compositional temporal networks, which make use of two or more features in their representation. After describing the temporal networks in detail, we will introduce the complexity classes that make up the polynomial-time hierarchy, as they will be useful in categorizing the complexity of particular controllability classes, before providing the appropriate complexity results.

A.1.1 Base Temporal Networks

We start by considering the different types of temporal networks that add conditional and disjunctive constraints to STNs. These networks have been well-studied and have corresponding completeness results associated with determining their feasibility. They will provide an appropriate background when we consider the effect of augmenting these networks with temporal uncertainty.

Disjunctive Networks

The first modification we make to STNs is to allow for disjunctions over temporal constraints. In practice, we frequently construct and consider schedules with disjunctive constraints; during a trip to the beach, we know that we want to eat lunch either 30 minutes before swimming or immediately afterwards – not at any moment in between.

The two types of disjunctive networks that are used in practice, Temporal Constraint Satisfaction Problems (TCSPs) and Disjunctive Temporal Networks (DTNs), differ in terms of the types of disjunctive constraints that they admit [22, 49].

Definition A.1. TCSP [22]

A TCSP is a 2-tuple $\langle X, R \rangle$ where:

- X is a set of event variables, whose domains are the reals
- R is a set of simple disjunctive constraints of the form $x_r - y_r \in \bigcup_k [l_{r,k}, u_{r,k}]$, where $x_r, y_r \in X$ and $l_{r,k}, u_{r,k} \in \mathbb{R}$

Definition A.2. DTN [49]

A DTN is a 2-tuple $\langle X, R \rangle$ where:

- X is a set of event variables, whose domains are the reals

- R is a set of full disjunctive constraints of the form $\bigvee_{k} (l_{r,k} \leq x_{r,k} - y_{r,k} \leq u_{r,k})$, where $x_{r,k}, y_{r,k} \in X$ and $l_{r,k}, u_{r,k} \in \mathbb{R}$

The disjunctive constraints of TCSPs require that every constraint in a given disjunction relates the same pair of events. In contrast, DTNs allow disjunctive constraints to be a disjunction over any constraints that might be found in an STN. In this thesis, we will refer to the type of disjunctions allowed by TCSPs as *simple disjunctions* and the type of disjunctions allowed by DTNs as *full disjunctions.* Checking the feasibility of both TCSPs and DTNs is known to be NP-complete [22, 49]. It is worth noting that a linear time transformation exists that converts DTNs into equivalent TCSPs [45], but maintaining the distinction between the two is important because, remarkably, as we extend the two types of networks, we see that the computational complexity of solving them will diverge.

Conditional Networks

The Conditional Simple Temporal Network (CSTN) represents a different way to augment STNs [54]. CSTNs allow for the introduction and observation of uncontrollable events and the conditional enforcement of constraints based on the observations of those events.

Definition A.3. CSTN [54]
A CSTN is a tuple $\langle X, R, P, O \rangle$ where:

- X is a set of event variables, whose domains are the reals

- R is a set of constraints of the form $\psi_r \rightarrow (l_r \leq x_r - y_r \leq u_r)$, where $x_r, y_r \in X$, ψ_r is a label representing a conjunction of propositions or their negations, and $l_r, u_r \in \mathbb{R}$

- P is a set of propositions

- O is a function mapping propositions in P to the events where their values are observed

To illustrate the usefulness of CSTNs, we provide an example. If we want to schedule the delivery of a package, we may prefer to use a CSTN to encode the urgency of the request; a package that we see marked as urgent, may need to be scheduled in the next 24 hours, but a package that is not marked as such can use a more relaxed schedule that guarantees shipment within the next week. Given event A representing when the package goes out for delivery and event B representing when the package must be delivered, we can encode the urgency using two constraints, if the package is urgent, we have the constraint $B - A \leq 1d$ with label u, and if the package is not urgent, we have the constraint $B - A \leq 7d$ with label $\neg u$.

What makes scheduling over CSTNs notable is that we may learn about the value of proposition u, or in this case the urgency of the package, at some unrelated event C that may differ from the events associated with the constraints they affect. In our given example, C represents the time at which the customer tells us the package's urgency. It is possible that the customer indicates that the package is urgent the day before dropping it off, but it is equally possible that the customer tells us the package is urgent several hours after they have already dropped it off. We conditionally enforce labeled constraints by observing the realized values of the propositions and checking whether a constraint's label, ψ_r is true. In the package example, we know that we will only need to enforce one of the two constraints based on what the observed value of u is at event C. We use the function O to encode the events at which specific propositions are observed.

Importantly, the true values of propositions are not "scheduled" in the same way that events are. Different instantiations of the same problem may yield different values for the propositions and, correspondingly, result in different constraints that must be enforced during execution. As a result, the scheduling problem for CSTNs is different than the one for STNs, TCSPs, and DTNs. In the previously described temporal networks, we knew the full set of constraints that would be enforced prior to scheduling and as such could satisfy all constraints with an implicitly static schedule. However, with CSTNs, there is no predetermined guarantee about when the scheduler learns about propositions, as the scheduler may have to predetermine a schedule that is robust to any learned proposition values or may have the flexibility to adapt the schedule on the fly. Across these different situations, different decisions may be made with respect to scheduling that may trade off between learning the actual values of propositions early in execution and maintaining a buffer of temporal flexibility. As such, when checking feasibility of CSTNs, we use *strong*, *weak*, and *dynamic consistency* to denote the different models under which the scheduler is guaranteed to learn the actual proposition values [54]. These models of consistency are analogous to the different notions of controllability that we consider in STNUs.

Strong consistency implies there exists a schedule that can be constructed that assigns values to all events in X, such that for every realization of the values of the propositions in P, all constraints in R are satisfied. Strong consistency checking of a CSTN reduces to checking the temporal consistency of the underlying STN and so is computable in $O(mn)$ time [54]. A CSTN is weakly consistent if for every assignment of values to the propositions in P, there exists some schedule can be constructed assigning values to event variables in X, such that all constraints in R are satisfied. Weak consistency checking of CSTNs is coNP-complete [54]. Dynamic consistency is concerned with whether it is possible to dynamically construct a schedule where assignment to values in X happen in order of event values and the true values of propositions $p \in P$ are learned only when the corresponding event given by $O(p)$ is executed. Dynamic consistency checking in CSTNs is PSPACE-complete [14].

A.1.2 Compositional Temporal Networks

We now provide definitions for the temporal networks that result when we combine conditional constraints, disjunctive constraints, and the consideration of temporal uncertainty.

Disjunctions and Temporal Uncertainty

We start by adding disjunctions to STNUs. As was the case with disjunctions added to STNs, when considering disjunctive temporal networks with uncertainty, we consider the effects of allowing both simple and full disjunctions.

Temporal Constraint Satisfaction Problems with Uncertainty (TCSPUs) augment STNUs by adding simple disjunctions over constraints.

Definition A.4. TCSPU [56]
A TCSPU is a 4-tuple $\langle X_e, X_c, R_r, R_c \rangle$ where:

- X_e is the set of executable events
- X_c is the set of contingent events
- R_r is the set of simple disjunctive temporal constraints over $X_c \cup X_e$
- R_c is the set of simple disjunctive contingent constraints

By augmenting a TCSPU with full disjunctions over temporal constraints, we get Disjunctive Temporal Networks with Uncertainty (DTNUs) [57].

Definition A.5. DTNU [44]
A DTNU is a 4-tuple $\langle X_e, X_c, R_r, R_c \rangle$ where:

- X_e is the set of executable events
- X_c is the set of contingent events
- R_r is the set of full disjunctive temporal constraints over $X_c \cup X_e$
- R_c is the set of simple disjunctive contingent constraints

It is worth noting that for DTNUs, all disjunctive contingent constraints are simple. Most models of temporal uncertainty assume that the duration of a contingent constraint is independent of any action taken by the scheduler. Accordingly, allowing disjunctive constraints to span different contingent constraints or to span contingent and requirement constraints would violate the spirit of this approach.

The concepts of strong, weak, and dynamic controllability as defined for STNUs scale immediately to temporal networks with disjunctions. However, the introduction of disjunctions makes the act of computing controllability much more difficult. The best available algorithms for deciding strong controllability of temporal networks with uncertainty and disjunction are in EXPSPACE [44]. Dynamic and weak controllability of these networks can be computed in EXPTIME, but these approaches also use exponential space. It is unknown whether any form of controllability checking for DTNUs or TCSPUs can be done in polynomial space [16, 57].

Conditions and Temporal Uncertainty

Extending STNUs instead with conditional constraints gives us Conditional Simple Temporal Networks with Uncertainty (CSTNUs) [30].

Definition A.6. CSTNU [18]
A CSTNU is a tuple $\langle X_e, X_c, R_e, P, O\rangle$ where:

- X_e is a set of executable events
- X_c is a set of contingent events
- R_r is a set of requirement constraints of the form $\psi_r \rightarrow (l_r \leq x_r - y_r \leq u_r)$, where $x_r, y_r \in X_e \cup X_c$, ψ_r is a label representing a conjunction of propositions or their negations, and $l_r, u_r \in \mathbb{R}$
- R_c is a set of contingent constraints of the form $0 \leq l_r \leq c_r - e_r \leq u_r$, where $c_r \in X_c, e_r \in X_e$ and $l_r, u_r \in \mathbb{R}$
- P is a set of propositions
- O is a function mapping propositions in P to the events where their values are observed

With CSTNUs, we now have two sources of external uncertainty, the observed values of propositions and the realized durations of contingent constraints. While we could evaluate consistency and controllability conditions separately (e.g. checking whether a network is strongly consistent while being dynamically controllable), we typically consider the two jointly. In other words, we assume that both the durations of contingent constraints and the values of the propositions are either never observed, all observed before execution, or observed along the way when we evaluate strong, weak, and dynamic controllability, respectively. Dynamic controllability of CSTNUs belongs to EXPTIME [16], but the complexity of checking strong and weak controllability are still open questions.

Conditions, Disjunctions, and Temporal Uncertainty

Finally, we combine conditions, disjunctions, and temporal uncertainty in a single network to get Conditional Disjunctive Temporal Networks with Uncertainty (CDTNUs).

Definition A.7. CDTNU

A CDTNU is a tuple $\langle X_e, X_c, R_e, P, O\rangle$ where:

- X_e is a set of executable events
- X_c is a set of contingent events
- R_e is a set of requirement constraints of the form $\bigvee_k \psi_{r,k} \rightarrow (l_{r,k} \leq x_{r,k} - y_{r,k} \leq u_{r,k})$, where $x_{r,k}, y_{r,k} \in X$, $\psi_{r,k}$ is a label representing a conjunction of propositions or their negations, and $l_{r,k}, u_{r,k} \in \mathbb{R}$
- R_c is a set of simple disjunctive contingent constraints
- P is a set of propositions
- O is a function mapping propositions in P to the events where their values are observed

We can apply the same techniques as those found in CSTNUs and DTNUs to show that dynamic controllability of CDTNUs can be computed in EXPTIME [16]. Algorithms for strong and weak controllability of CDTNUs have not yet been developed.

A.1.3 Polynomial Time Hierarchy

Before we continue to the actual complexity results it is useful to briefly introduce the polynomial-time hierarchy [50], as it will allow us to more precisely characterize the difficulty of some of our controllability problems.

The classes Σ_k^P and Π_k^P are defined recursively. We start with $\Sigma_1^P = \text{NP}$ and $\Pi_1^P = \text{coNP}$ and define Σ_{k+1}^P as $\text{NP}^{\Sigma_k^P}$ and Π_{k+1}^P as $\text{coNP}^{\Sigma_k^P}$, where A^B represents the set of problems that can be solved in complexity class A if an oracle for a B-complete problem is provided.

In this appendix, we will pay close attention to the complexity classes Σ_2^P and Π_2^P and will make heavy use of the fact that $\Sigma_k^P = \text{co}\Pi_k^P$ and that $\forall\exists$3SAT is a Π_2^P-complete problem, where $\forall\exists$3SAT is the problem of determining whether for a given 3-CNF $\Phi(\vec{x}, \vec{y})$ it is the case that for all $\vec{y}$, there exists $\vec{x}$, such that $\Phi(\vec{x}, \vec{y})$ is true [50]. Σ_k^P and Π_k^P are also known to be fully contained within PSPACE, meaning that membership to any complexity class in the polynomial-time hierarchy guarantees the existence of a deterministic algorithm that uses at most polynomial space.

A.1.4 Evaluating Complexity

While complexity results for the base temporal networks we have described are well-known, very few tight bounds exist for the networks derived from their composition, despite the fact that much work has been done to develop algorithms for them. Many of their hardness lower-bounds can be inherited from the base temporal networks, but it is an open question whether or not they are tight.

In this subsection, we will prove complexity class completeness results for each of strong, weak, and dynamic controllability for each network, updating the hardness lower-bounds as needed before demonstrating membership to the appropriate class. When describing the controllability decision problems, we will use the prefixes SC-, WC-, and DC- to refer to checking the strong, weak, and dynamic controllability of the denoted temporal network, respectively.

Hardness Results

We start by providing tighter hardness lower-bounds for the controllability problems across temporal networks. Existing results for CSTNs give us appropriate lower-bounds for CSTNUs, but for the temporal networks with disjunction and uncertainty, we need tighter analysis than the NP-hardness provided by TCSPs and DTNs.

Lemma A.1. *Checking the weak controllability of a TCSPU is Π_2^P-hard.*

Proof. To show WC-TCSPU is Π_2^P-hard, we will provide a reduction from $\forall\exists$3SAT. In other words, we want to construct a TCSPU T such that a formula $\forall\vec{y}, \exists\vec{x} : \phi(\vec{x}, \vec{y})$ is weakly controllable if and only if T is weakly controllable, where $\vec{x}, \vec{y}$ are vectors of boolean values, and ϕ is a 3-CNF formula.

We start by defining our events, starting with a reference event Z. For each x_i, we construct event t_{x_i} with disjunctive constraint $t_{x_i} - Z \in [0, 0] \cup [1, 1]$. For each y_j, we also construct event t_{y_j} with contingent constraint $t_{y_j} - Z \in [0, 0] \cup [1, 1]$. These events will represent the initial values chosen against which we will evaluate ϕ with 0 corresponding to an assignment of false and 1 corresponding to true.

For convenience, we also add events corresponding to the negations of each variable. $t_{\overline{x}_i}$ has two corresponding constraints, $t_{\overline{x}_i} - Z \in [0, 0] \cup [1, 1]$ and $t_{\overline{x}_i} - t_{x_i} \in [-1, -1] \cup [1, 1]$. This ensures that $t_{\overline{x}_i}$ takes on a different value than t_{x_i}. Similarly, we add new events $t_{\overline{y}_j}$ with requirement constraints $t_{\overline{y}_j} - Z \in [0, 0] \cup [1, 1]$ and $t_{\overline{y}_j} - t_{y_j} \in [-1, -1] \cup [1, 1]$. We will rely on the fact that we are evaluating weak controllability to ensure that we set the events for the negated variables in response to the values assigned by nature.

We now move on to encoding each individual clause of ϕ into our TCSPU T. Our approach is going to be highly inspired by the reduction from 3SAT to the 3-coloring problem on graphs and the reduction from 3-coloring to computing feasibility of a TCSP [22]. We will emulate the three colors by requiring all events to occur at time 0, 1, or 2 and enforce that two nodes t_i, t_j differ in value by requiring that $t_i - t_j \in \{-2, -1, 1, 2\}$.

For each clause c_k of ψ, we create a new gadget whose output represents the truth

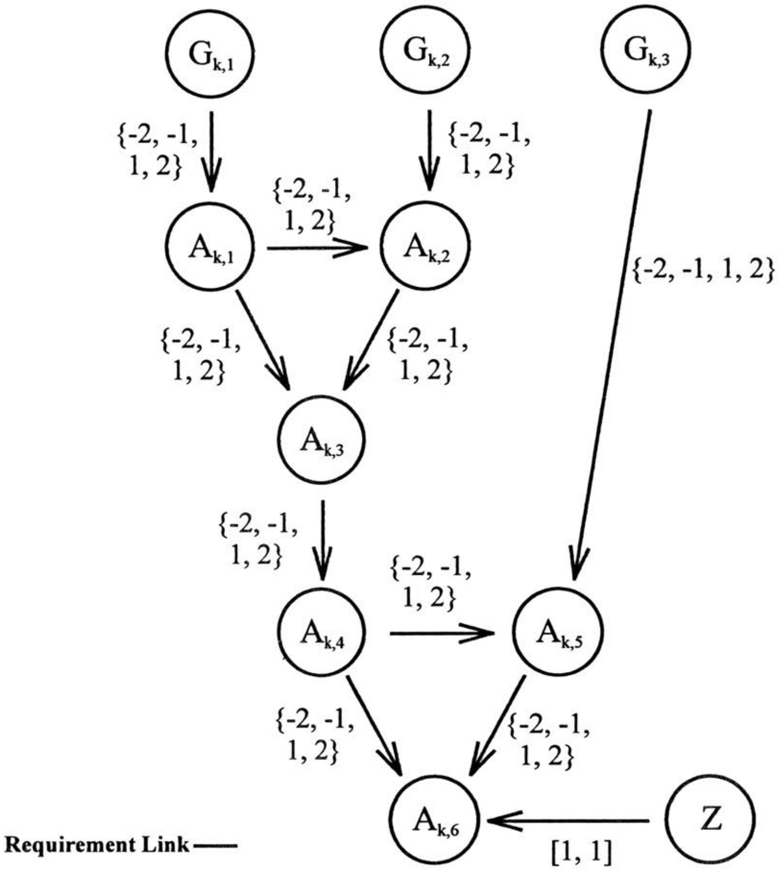


Figure A-2: A gadget used in the proof that WC-TCSPU is Π_2^P-hard. The A_k events can each take on any value from $\{0, 1, 2\}$. The value $A_{k,6}$ represents the disjunction of $G_{k,1}, G_{k,2}, G_{k,3}$ and is constrained to equal one.

value of c_k (see Figure A-2). Each event $G_{k,l}$ represents the truth value of literal l of clause c_k. We require that the value matches the initially assigned value of literal q by adding the constraint $G_{k,l} - t_q = 0$. The layout of events A_k weakly emulate an or-gate, where $A_{k,6}$ is the output and constrained to have a value of 1. For any values of the events G_k, it is possible to assign all of the events A_k such that $A_{k,6} = 1$ except for when $G_{k,1} = G_{k,2} = G_{k,3} = 0$. As a result, it is possible to choose a set of values for the events to satisfy the constraints of the gadget so long as at least one literal of the original clause c_k is true.

Taken together, if there exists an assignment of values to events such that each gadget's constraints are satisfied, then for whichever particular $\vec{y}$ we start with, then

$\exists\vec{x} : \phi(\vec{x}, \vec{y})$. When checking weak controllability, all executable events are assigned values after the contingent events, so as we have constructed it, T is weakly controllable if and only if $\forall\vec{y}, \exists\vec{x} : \phi(\vec{x}, \vec{y})$. Thus, WC-TCSPU is Π_2^P-hard. □

Lemma A.2. *Checking the dynamic controllability of a TCSPU is PSPACE-hard.*

Proof. To show that DC-TCSPU is PSPACE-hard, we provide a reduction from TQBF, which is known to be PSPACE-complete, to DC-TCSPU. In particular, for a problem of the form $\exists x_1 \forall y_1 ... \exists x_n \forall y_n : \phi(\vec{x}, \vec{y})$, we construct a TCSPU T such that T is dynamically controllable if and only if $\exists x_1 \forall y_1 ... \exists x_n \forall y_n : \phi(\vec{x}, \vec{y})$, where ϕ is a 3-CNF formula.

Ideally, we would employ a strategy similar to our transformation for WC-TCSPU in Lemma A.1, but in that construction, many of the clausal gadget events can occur before the contingent events they relate to are assigned by nature. Because dynamic controllability requires events to be assigned reactively in a just-in-time manner, we must make sure that all values of $\vec{y}$ are encoded and specified by the network before we do any subsequent computation.

We start by encoding the alternating choice of x_i and y_i as represented by the values decided by the scheduler and nature. We start with an anchor point O and for each x_i and y_i, we create events $\tau_{x_i,s}$, $\tau_{x_i,e}$, $\tau_{y_i,s}$, and $\tau_{y_i,e}$. For each x_i, we create a requirement constraint of $\tau_{x_i,e} - \tau_{x_i,s} \in [0,0] \cup [1,1]$, and for each y_i, we create a contingent constraint of $\tau_{y_i,e} - \tau_{y_i,s} \in [0,0] \cup [1,1]$. This enforces that the difference between the start and end values is either 0 or 1, corresponding to an assignment of false or true in the original formula. To ensure that the values are chosen in order when evaluated in a dynamic controllability setting, we require that $\tau_{x_i,s} - O = 2i-2$ and that $\tau_{y_i,s} - O = 2i-1$. This gives us the exact alternating pattern as described by the original formula, and what remains is to evaluate the truth condition.

Our strategy for evaluating the truth of the formula is to replicate the same structures used by the constructed TCSPU in Lemma A.1. We create a secondary anchor point Z with $Z - O = 2n+2$ to ensure that Z happens after all boolean values have been assigned, and then create new events corresponding to the values of $\vec{x}$ and

$\vec{y}$ that are anchored at Z instead of at different times during the execution. For each x_i, we create t_{x_i} with the constraint $t_{x_i} - \tau_{x_i,e} = 2(n-i)+4$, and for each y_i, we create t_{y_i} with the constraint $t_{y_i} - \tau_{y_i,e} = 2(n-i)+3$. The rest of the construction, namely the construction of the negated literal values and the clausal gadgets, remains the same, and by the same reasoning, we see that it is possible for a given assignment, it is possible for all constraints to be respected if and only if ϕ is satisfied by that assignment of values. Since the initial events are set up such that when the entire network is dynamically controllable the values of events are chosen in the same order as the quantification of the original TQBF formula, we know that T is dynamically controllable if and only if $\exists x_1 \forall y_1 ... \exists x_n \forall y_n : \phi(\vec{x}, \vec{y})$. Because the new network can be constructed in polynomial time, we have a polynomial time reduction from TQBF to DC-TCSPU, so DC-TCSPU is PSPACE-hard.

□

Lemma A.3. *Checking the strong controllability of a DTNU is Σ_2^P-hard.*

Proof. To prove that SC-DTNU is Σ_2^P-hard, we will reduce the complement of $\forall\exists$3SAT, a Π_2^P-complete problem, to SC-DTNU.

An example problem of $\forall\exists$3SAT is of the form $\forall\vec{x}, \exists\vec{y} : \phi(\vec{x}, \vec{y})$, where $\vec{x}, \vec{y}$ are vectors of boolean values and ϕ is a 3-CNF formula. The complementary problem is $\exists\vec{x}, \forall\vec{y} : \psi(\vec{x}, \vec{y})$, where ψ is a 3-DNF formula representing the negation of ϕ. Given the input problem, we construct a corresponding DTNU D that is strongly controllable if and only if the complementary formula ψ is true (if the original formula ϕ is false).

First we define the events of D. We start with a reference event Z, which represents the first point to be executed. For each $x_i \in \vec{x}$, we add points t_{x_i} and $t_{\overline{x}_i}$ to represent the value of x_i and its negation during some candidate assignment to our formula. We do the same thing for $\vec{y}$ adding t_{y_j} and $t_{\overline{y}_j}$ for each $y_j \in \vec{y}$. We also introduce a new gadget per clause of ψ (see Figures A-3 and A-4) and in each gadget, we introduce ten new events. Events $G_{k,1}$, $G_{k,2}$, and $G_{k,3}$ represent the values of each literal of clause k and event $G_{k,and}$ represents the value of the conjunction of those literals. For each clause, we also add $A_{k,1}$, $A_{k,2}$, $A_{k,3}$, $A_{k,4}$, $A_{k,5}$, and $A_{k,6}$ which are used collectively to simulate an and clause. By appropriately adding contingent and requirement constraints between these events, we will get a DTNU that is controllable if and only if the original formula ψ is true.

We start by adding constraints to encode the initial assignment of values. For each t_{x_i} we add a simple disjunctive constraint requiring that $t_{x_i} - Z \in [0,0] \cup [1,1]$. Similarly, for each t_{y_j}, we add a disjunctive *contingent* constraint enforcing $t_{y_j} - Z \in [0,0] \cup [1,1]$. The choice of values for these initial events maps directly back to an assignment of values in the 3-DNF formula ψ with 0 representing false and 1 representing true.

We also enforce the values of the negations of these variables for convenience, with the same simple disjunctive constraint requiring $t_{\overline{x}_i} - Z \in [0,0] \cup [1,1]$ and the disjunctive contingent constraint enforcing $t_{\overline{y}_j} - Z \in [0,0] \cup [1,1]$. To ensure that x_i and its negation take on values we also add the requirement that $t_{x_i} - t_{\overline{x}_i} \in \cup[-1,-1] \cup [1,1]$. We will discuss our strategy for ensuring that the values of t_{y_j} and $t_{\overline{y}_j}$ differ below.

We now move on to the constraints associated with the clausal gadgets. $G_{k,l}$ represents the truth value of the l^{th} element of clause k, and $G_{k,and}$ represents the truth value of the entire clause; each event, $G_{k,*}$, that is newly created for the gadget is initialized using a contingent constraint enforcing $G - Z \in [0,0] \cup [1,1]$. We also create a disjunctive constraint across all gadgets, such that if for any k, $G_{k,and} - Z = 1$, then the constraint is satisfied. We call this disjunctive constraint the *goal constraint.* This has an immediate correspondence to the notion that the entire formula ψ is satisfied if any of its constituent clauses is satisfied.

Our current construction makes heavy use of contingent constraints, and while we may want the events in our gadgets to represent certain values, their values are chosen by nature, meaning we have no way to directly control their values.

However, we do have control over the constraints of D and, in particular, the disjunctive constraint that spans the gadgets. Checking strong controllability can be seen as a two-player game, where the scheduler goes first and nature goes second. Nature's goal is to construct an assignment such that some constraint is violated.

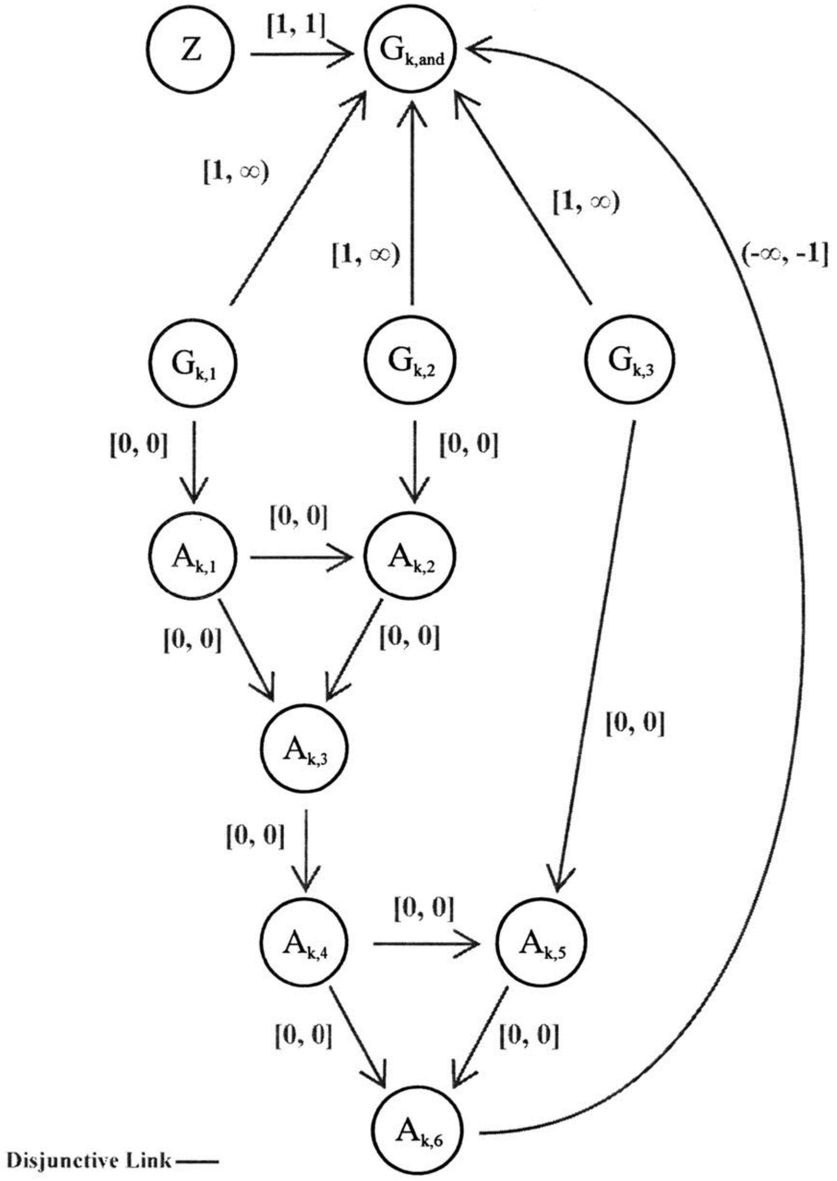


Figure A-3: A description of the disjunctive goal constraints found in each gadget used in the proof that SC-DTNU is Σ_2^P-hard. The A_k event can each take on any value from $\{0, 1, 2\}$. The value $A_{k,6}$ will only be precluded from taking on a value of 0 when all of $G_{k,1}, G_{k,2}, G_{k,3}$ are 1. The disjunctive constraints of this gadget are all individual parts of the larger collective disjunctive goal constraint.

Upon closer examination, we see that in our construction, the only constraint that can be affected by the contingent constraint durations chosen by nature is the goal constraint. If there exist certain combinations of contingent constraint durations that we want to preclude from our evaluation, we preclude them by adding additional

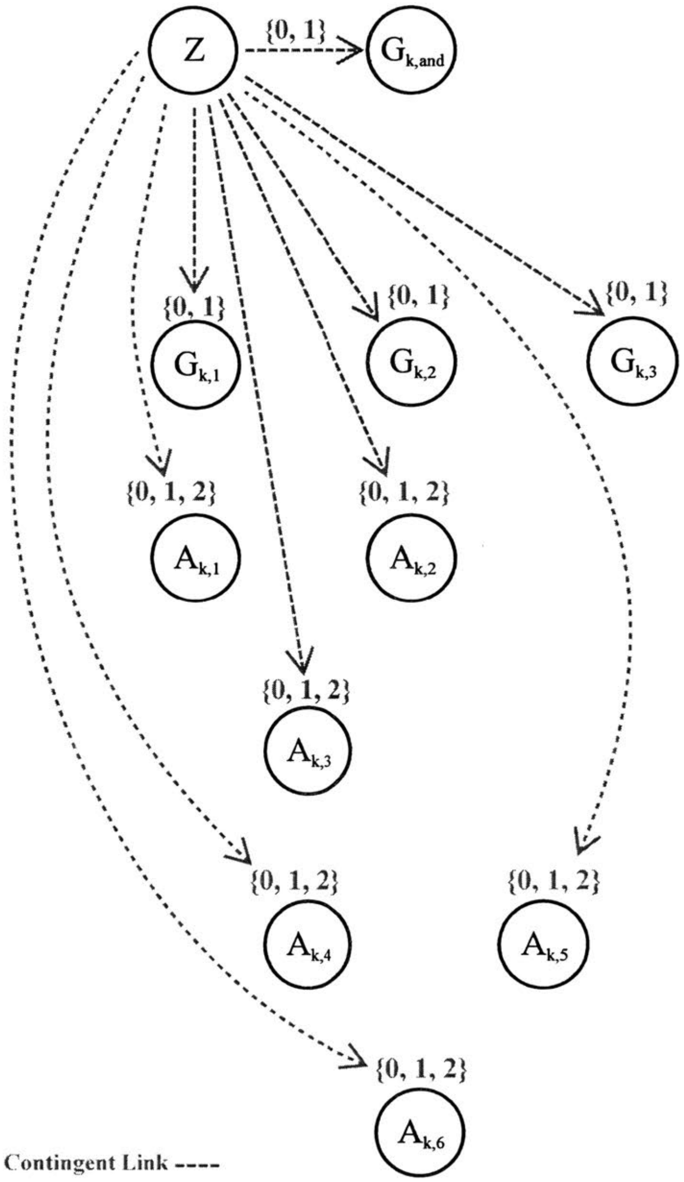


Figure A-4: A description of the contingent constraints found in each gadget used in the proof that SC-DTNU is Σ_2^P-hard. The constraints between Z and each $G_{k,l}$ are contingent constraints but are constrained to be equal in length to the original x_i, y_j they relate to using the shared disjunctive goal constraint.

disjunct to the goal constraint that are satisfied when those contingent constraints take on those durations. In this way, any contingent constraint values that do not conform to our desired structure make D trivially controllable, and controllability then reduces to controllability under our desired set of constraints across contingent constraints.

First, we need to make sure that the events t_{y_j} and $t_{\overline{y}_j}$ take on different values. We ensure this by adding $t_{y_j} - t_{\overline{y}_j} = 0$ to our goal constraint; if nature gives t_{y_j} and $t_{\overline{y}_j}$ the same value, then we trivially ignore this case. Similarly, since we want $G_{k,l}$ to take on the same value as the literal q it represents, we augment our disjunctive goal constraint with $G_{k,l} - t_q \in [-1,-1]$ and $G_{k,l} - t_q \in [1,1]$ where t_q is the event associated with literal q. As a result, if the clausal representation of the variable differs from our assignment, our network is trivially controllable.

We enforce the conjunction of the elements of each clause by augmenting our goal constraint with $G_{k,and} - G_{k,l} \geq 1$ for each $G_{k,l}$ of our clause gadget. Since each event of our gadget can take on a value of 0 or 1, this constraint will only be satisfied if some literal value is 0 while $G_{k,and}$ has a value of 1. In these situations, $G_{k,and}$ does not represent the conjunction of the literals of clause k, and our network then becomes trivially controllable.

Unfortunately, our network still does not perfectly encode the conjunction seen in a DNF clause. It is possible for each $G_{k,l}$ to take on a value of 1 while $G_{k,and}$ is assigned a value of 0. As a result, it may be the case that the original problem, $\exists\vec{x}\forall\vec{y}\psi(\vec{x},\vec{y})$ is true but each $G_{k,and}$ is set to 0, meaning that the network is not strongly controllable.

To fix this, we must augment our gadget to enforce that identical inputs have the same output. This is the reason for introducing events $A_{k,m}$, and these events' values are set by new contingent constraints that enforce $A_{k,m} - Z \in [0,0] \cup [1,1] \cup [2,2]$. Through an exhaustive enumeration of possible values, we can confirm that whenever $G_{k,1}, G_{k,2}, G_{k,3}$ are all 1, either $A_{k,6}$ will be 1 or one of the disjuncts of the goal constraints (see Figure A-3) will be satisfied. In this case, when we add $A_{k,6} - G_{k,and} \geq 1$ to the goal constraint, we know that when $G_{k,1}, G_{k,2}, G_{k,3}$ are all equal to 1, D is controllable, as either $G_{k,and} = 1$, meaning $G_{k,and} - Z = 1$, which satisfies the goal constraint, or $G_{k,and} = 0$, implying $A_{k,6} - G_{k,and} = 1$, which also satisfies the goal constraint.

Before continuing, we need to confirm that the addition of the new sub-gadget does not introduce any new problems. For all other values of $G_{k,1}$, $G_{k,2}$, and $G_{k,3}$, we know it is possible for $A_{k,6}$ to take on a value of 0. Since $A_{k,6}$ is the only event of the or-gate that is related to other values by the goal constraint and setting it to 0 does not satisfy the goal constraint, we know that if $G_{k,1}, G_{k,2}, G_{k,3}$ are not all 1, then there exists a choice of values by nature such that the goal constraint is not satisfied by gadget k.

Our transformation is complete and because there is one gadget per clause in ψ and each gadget is of constant size, we see that the transformation takes polynomial time. What remains is to show that D is strongly controllable if and only if $\exists\vec{x}\forall\vec{y}\psi(\vec{x},\vec{y})$ is true. This is evident from our construction.

If D is strongly controllable, there must be some set of assignment to values t_{x_i} such that no possible assignment of values to the other events violates any of the constraints. We prove this by contradiction, assuming that although our choice of t_{x_i} guarantees the satisfaction of all other constraints in D, there is no choice of $\vec{x}$ that guarantees satisfaction of $\forall\vec{y}\psi(\vec{x},\vec{y})$. Let $\vec{x}$ be specified such that x_i is true if and only if $t_{x_i} = 1$. If ψ is not guaranteed to be satisfied, there must be some $\vec{y}$ such that $\psi(\vec{x},\vec{y})$ is false. Returning to D, assume that nature specifies t_{y_j} such that $t_{y_j} = 1$ if and only if y_j is true. Since D is strongly controllable, we know that some disjunctive goal constraint is satisfied no matter the assignment of contingent event variables. Let's assume that all $t_{\overline{y}_j}$ are chosen such that they represent the negation of their corresponding t_{y_j}, that all $A_{k,m}$ of the gadgets are chosen such that the disjunctive constraints involved between all $G_{k,l}$ and $A_{k,m}$ are not satisfied, and that all $G_{k,and}$ are chosen to be 0. The only remaining disjunctive constraints are those involving each $G_{k,and}$. For any particular k, setting $G_{k,and}$ to 0 only satisfies a constraint if $A_{k,6}$ is 1, so given all these assumptions, at least one $A_{k,6}$ must be set to 1 (otherwise the system would be uncontrollable). As we demonstrated earlier, $A_{k,6}$ is only constrained to be 1 when all of $G_{k,1}$, $G_{k,2}$, and $G_{k,3}$ are also 1. But those three values correspond exactly to literals in a clause of $\psi(\vec{x},\vec{y})$. If all three are 1, then we have a true clause and because ψ is a 3-DNF formula, this means that ψ is true. We have a contradiction. Therefore if D is controllable, $\exists\vec{x}\forall\vec{y}\psi(\vec{x},\vec{y})$.

To conclude we show the reverse direction, that if $\exists\vec{x}\forall\vec{y}\psi(\vec{x},\vec{y})$ is true, then D is strongly controllable. Let $\vec{x}$ be the assignment of variables that guarantees $\forall\vec{y}\psi(\vec{x},\vec{y})$;

we show how to use $\vec{x}$ to show that D is strongly controllable. We pick our t_{x_i} such that $t_{x_i} = 1$ if and only if x_i is true and will pick our $t_{\overline{x}_i}$ such that $t_{\overline{x}_i} \neq t_{x_i}$. Again, we proceed with proof by contradiction, assuming that D is not strongly controllable. Our choice of t_{x_i} and $t_{\overline{x}_i}$ satisfy all constraints except the disjunctive goal constraint, so there must be a choice of contingent events that violate the disjunctive goal constraint. We know setting $G_{k,and} = 1$ satisfies the goal constraint, so all $G_{k,and} = 0$. By proxy, for all k, $A_{k,6} = 0$ to ensure that the goal constraint is not satisfied because of the constraint between $A_{k,6}$ and $G_{k,and}$. Because $A_{k,6} = 0$, it must be the case that for each gadget, at least one of $G_{k,1}$, $G_{k,2}$, or $G_{k,3}$ must equal zero. In order for the goal disjunctive constraint to remain unsatisfied, each $G_{k,l}$ must maintain the same value as some t_{x_i}, $t_{\overline{x}_i}$, t_{y_j}, or $t_{\overline{y}_j}$ based on the values of the clauses of ϕ. This forces a particular assignment of values to t_{y_j} which can be mapped back to some $\vec{y}$. For that particular assignment, we know that $\psi(\vec{x}, \vec{y})$ is true, or that there is some clause k' with all literals set to true. This contradicts the fact that for all k, at least one of $G_{k,1}$, $G_{k,2}$, or $G_{k,3}$ must be zero. Thus, D must be strongly controllable, and we have proven that SC-DTNU is Σ_2^P-hard.

□

Completeness

We now move on to proving completeness for the controllability problems on each temporal network. Our general approach for characterizing the complexity of controllability problems will be to map an inputted temporal network to a corresponding system of conditional linear inequalities that encode the same constraints. We will then use existential and universal quantifiers over the variables to dictate which type of controllability is being determined.

Our transformation proceeds as follows. We imagine the execution of a temporal network as being a game played between two agents, the scheduler and nature, where the scheduler assigns times to executable events and nature assigns times to contingent events. In general the question of determining controllability will reduce to the problem of evaluating a quantified linear system and our techniques draw inspiration from and are related to approaches in those areas [24, 51].

For notational convenience, we will split our variables into $\vec{x}$ and $\vec{y}$ for those assigned by the scheduler and nature, respectively. For each executable event e_i, we create a new variable x_i, and for each contingent event c_i, we create a new variable y_i.

We create a one-to-one mapping between the set of temporal network constraints and the new linear inequalities. First, we replace all executable events e_i with the corresponding x_i. With the contingent events, however, we need to be more careful. For each contingent event c_i, we find the contingent constraint that restricts it of the form $l_c \leq c_i - e_j \leq u_c$. We then replace each instance of c_i in our constraints with $y_i + x_j$. Our reason for doing this has to do with the nature of contingent constraints. In temporal networks, there is a guarantee that nature respects the contingent constraint bounds in relation to its corresponding starting executable event. So while free constraints relate events in terms of the absolute time of their occurrence, contingent constraints require nature to respect relative timings of events. If the durations of contingent constraints are to be known before scheduling begins, as is the case with weak controllability, then the constructed system of linear inequalities will fail to map to the base temporal network if nature is asked to pick the precise times of contingent events.

After the substitutions, each constraint is a combination of conditional linear inequalities of the form $\psi \rightarrow \vec{a} \cdot \begin{bmatrix} \vec{x} \\ \vec{y} \end{bmatrix} \leq b$, where b is some constant, ψ is a (possibly empty) precondition for the enforcement of the constraint, and $\vec{a}$ represents the coefficients of the constraints where each coefficient is either -1, 0, or 1. Since all constraints are relative, without loss of generality, we specify that the earliest event happens at time $t = 0$, meaning we can safely require that $\vec{x} \geq 0$. When we quantify over variables to pick controllability, we require that each x_i has an existential quantifier and each y_i has a for-all quantifier drawn from the union of the ranges $[l_1, u_1], ..., [l_d, u_d]$, where l_j and u_j are retrieved from one of c_i's corresponding contingent constraints.

When evaluating controllability for disjunctive networks, it is useful to consider each contingent range separately, and so we will define Ω as a mapping from each variable y_i and one of its possible continuous ranges. In general, we will use the shorthand $\forall\Omega$ to indicate that we are considering all possible mappings and $\forall\vec{y} \in \Omega$ to specify that we are drawing our $\vec{y}$ from one particular mapping. Our choice of the ordering of the quantifiers will dictate which type of controllability will be considered. We also must consider how conditions affect our model, and will define Ψ as the full set of conditions that can be observed by the scheduler when our temporal networks include conditional constraints.

It is also worth noting that whenever we consider a vector of values $\vec{x_c}$ that represent a solution to our scheduling problem, we assume that the representation of $\vec{x_c}$ is polynomial in the size of the original input. While we are agnostic to which particular representation is used, we do still require a fixed number of bits required to represent each individual number. The implication of this is that between any two numbers, there are a finite number of intermediate values that can be represented.

The rest of our analysis is divided into an analysis of strong controllability over temporal networks, weak controllability over temporal networks, and dynamic controllability over temporal networks.

Strong Controllability

Theorem A.4. *Checking strong controllability of a CSTNU is in P.*

Proof. We start by encoding the SC-CSTNU problem in our described format:

$$\exists\vec{x}\forall\vec{y}\forall\Psi : \bigwedge_i \psi_i \rightarrow \vec{a_i} \cdot \begin{bmatrix} \vec{x} \\ \\ \vec{y} \end{bmatrix} \leq b_i$$

Because $\forall \bigwedge \phi$ is the same as $\bigwedge \forall\phi$, we can rewrite our problem as:

$$\exists\vec{x}\forall\vec{y} \bigwedge_i \forall\Psi : \psi_i \rightarrow \vec{a_i} \cdot \begin{bmatrix} \vec{x} \\ \\ \vec{y} \end{bmatrix} \leq b_i$$

Since the inner equation must hold for all Ψ, it must also hold when ψ_i is true, allowing us to eliminate the conditionals:

$$\exists\vec{x}\forall\vec{y}\bigwedge_i : \vec{a_i}\cdot\begin{bmatrix}\vec{x}\\ \\ \vec{y}\end{bmatrix}\leq b_i$$

But of course, this is exactly the encoding for checking strong controllability of an STNU. Since STNU strong controllability is verifiable in polynomial time [58], our work demonstrates that strong controllability of a CSTNU can be determined in polynomial time through reduction to an STNU. □

Theorem A.5. *Checking the strong controllability of TCSPUs is NP-complete.*

Proof. We know that checking the feasibility of a TCSP is NP-hard [22], and because TCSPUs are a generalization of TCSPs, it follows that SC-TCSPU is NP-hard. To prove completeness, we show that SC-TCSPU $\in$ NP.

In a TCSPU, all disjunctive requirement constraints span the same pair of variables, meaning that every requirement constraint is of the form $t_i - t_j \in [l_1, u_1] \cup \ldots \cup [l_k, u_k]$, where for every $p < q$, $u_p < l_q$. This allows us to rewrite all individual constraints as:

$$t_i - t_j \geq l_1 \wedge \left(\bigwedge_{p=2}^{k} t_i - t_j \leq u_{p-1} \vee t_i - t_j \geq l_p\right) \wedge$$

$$t_i - t_j \leq u_k$$

Now when we encode strong controllability of a TCSPU, we can write the formula as:

$$\exists\vec{x}\forall\vec{y}: \bigwedge_i\bigvee_j \vec{a_{ij}}\cdot\begin{bmatrix}\vec{x}\\ \\ \vec{y}\end{bmatrix}\leq b_{ij}$$

$$\exists \vec{x} \bigwedge_i \forall \vec{y} : \bigvee_j \vec{a_{ij}} \cdot \begin{bmatrix} \vec{x} \\ \vec{y} \end{bmatrix} \leq b_{ij}$$

$$\exists \vec{x} \bigwedge_i \neg \exists \vec{y} : \bigwedge_j \vec{a_{ij}} \cdot \begin{bmatrix} \vec{x} \\ \vec{y} \end{bmatrix} > b_{ij}$$

For any fixed $\hat{x}$ and i, we can solve the problem $\exists \vec{y} : \bigwedge_j [\hat{x}^T; \vec{y}^T] \cdot \vec{a_{ij}} > b_{ij}$ in polynomial time. We know that linear programs can be optimized in polynomial time [32], and so to derive an answer for our original problem, we solve the linear program $\bigwedge_j [\vec{a_{ij}}^T; -1] \begin{bmatrix} \hat{x} \\ \vec{y} \\ \epsilon \end{bmatrix} \geq b_{ij}$ maximizing ϵ. If no solution exists, then there is no valid $\vec{y}$. If a solution exists with $\epsilon \leq 0$, then there was some constraint for which $[\hat{x}^T; \vec{y}^T] \cdot \vec{a_{ij}} > b_{ij}$ did not hold as there was a non-positive margin required to make all inequalities hold. Thus, only if $\epsilon > 0$, do we say that there exists a $\vec{y}$ satisfying our original constraints.

This immediately implies that we have a routine for verifying a certificate for SC-TCSPU in polynomial time. Given a certificate $\hat{x}$, then for each of the constraints i, we run our subroutine for determining whether a $\vec{y}$ exists that satisfies the specified sub-constraints. Since the verification algorithm runs in polynomial time, we know that SC-TCSPU $\in$ NP, and that SC-TCSPU is NP-complete. □

Theorem A.6. *Checking the strong controllability of DTNUs and CDTNUs are Σ_2^P-complete.*

Proof. We know that checking the strong controllability of a DTNU is Σ_2^P-hard from Lemma A.3 and because CDTNUs generalize DTNUs, SC-CDTNU is also Σ_2^P-hard. To demonstrate that both are Σ_2^P-complete, we show that checking the strong controllability of a CDTNU is in Σ_2^P.

To do so, we show that with an NP oracle we can verify that a CDTNU is strongly controllable in polynomial time. We start with an encoding of our problem:

$$\exists \vec{x} \forall \Psi \forall \Omega \forall \vec{y} \in \Omega : \bigwedge_i \bigvee_j \psi_{ij} \rightarrow \bigwedge_k \vec{a}_{ijk} \cdot \begin{bmatrix} \vec{x} \\ \\ \vec{y} \end{bmatrix} \leq b_{ijk}$$

and we let our certificate be the assignment of values to all executable events, $\hat{x}$. Given this certificate, an NP-oracle is capable of evaluating:

$$\exists \Psi \exists \Omega \exists \vec{y} \in \Omega : \neg \bigwedge_i \bigvee_j \psi_{ij} \rightarrow \bigwedge_k \vec{a}_{ijk} \cdot \begin{bmatrix} \hat{x} \\ \\ \vec{y} \end{bmatrix} \leq b_{ijk}$$

We can see this simply, as when we provide a certificate comprised of $\hat{\Psi}, \hat{\Omega}, \hat{y}$, it takes linear time to verify whether the conditional constraints are all satisfied.

Thus, when given a candidate assignment $\hat{x}$, we can use an NP-oracle to evaluate the negation of the remainder of the formula. If the negation has no solution, then we know that the original formula is true, and we have a way to verify SC-CDTNU in polynomial time. Thus, SC-CDTNU $\in \Sigma_2^P$, so SC-DTNU and SC-CDTNU are Σ_2^P-complete. □

Weak Controllability

Next, we move on to evaluating the complexity of weak controllability in temporal networks.

Theorem A.7. *Checking the weak controllability of CSTNUs is coNP-complete.*

Proof. Checking the weak controllability of STNUs is coNP-complete [39], so similarly checking the weak controllability of CSTNUs must be coNP-hard. To demonstrate that WC-CSTNU is coNP-complete, we must show that WC-CSTNU $\in$ coNP. We see this clearly when we look at the quantified linear system we get when evaluating a CSTNU's weak controllability:

$$\forall \Psi \forall \vec{y} \exists \vec{x} : \bigwedge_i \psi_i \rightarrow \vec{a}_i \cdot \begin{bmatrix} \vec{x} \\ \\ \vec{y} \end{bmatrix} \leq b_i$$

To show that WC-CSTNU is in coNP, we show that its complement problem is in NP, or that we can verify the following formula in polynomial time:

$$\exists \Psi \exists \vec{y} \neg \exists \vec{x} : \bigwedge_i \psi_i \rightarrow \vec{a}_i \cdot \begin{bmatrix} \vec{x} \\ \\ \vec{y} \end{bmatrix} \leq b_i$$

In this instance, we take as our certificate a particular choice of $\hat{\Psi}$ and $\hat{y}$. We can verify these values directly:

$$\neg \exists \vec{x} : \bigwedge_{i : \hat{\Psi} \models \psi_i} \psi_i \rightarrow \vec{a}_i \cdot \begin{bmatrix} \vec{x} \\ \\ \hat{y} \end{bmatrix} \leq b_i$$

$$\neg \exists \vec{x} : \bigwedge_{i : \hat{\Psi} \models \psi_i} \vec{a}_i \cdot \begin{bmatrix} \vec{x} \\ \\ \hat{y} \end{bmatrix} \leq b_i$$

Of course, we can evaluate all linear inequalities simultaneously through the evaluation of a linear program:

$$\neg \exists \vec{x} : A_{\hat{\Psi}} \begin{bmatrix} \vec{x} \\ \\ \hat{y} \end{bmatrix} \leq \vec{b}_{\hat{\Psi}}$$

Since linear programs can be evaluated in polynomial time [32], we can verify the complement of WC-CSTNU in polynomial time, meaning that WC-CSTNU $\in$ coNP and is coNP-complete. □

Theorem A.8. *Checking the weak controllability of TCSPUs, DTNUs, and CDTNUs are Π_2^P-complete.*

Proof. By Lemma A.1, we know that computing the weak controllability of TCSPUs are Π_2^P-hard, meaning computing WC-DTNU and WC-CDTNU are both also Π_2^P-hard. To show that all three are Π_2^P-complete, we must show that WC-CDTNU $\in \Pi_2^P$. We start with the quantified formula representation of weak controllability in a CDTNU:

$$\forall \Psi \forall \Omega \forall \vec{y} \in \Omega \exists \vec{x} : \bigwedge_i \bigvee_j \psi_{ij} \rightarrow \left(\bigwedge_k \vec{a}_{ijk} \cdot \begin{bmatrix} \vec{x} \\ \\ \vec{y} \end{bmatrix} \leq b_{ijk} \right)$$

For our purposes, it will be useful to show that the complementary problem is in Σ_2^P:

$$\exists \Psi \exists \Omega \exists \vec{y} \neg \exists \vec{x} : \bigwedge_i \bigvee_j \psi_{ij} \rightarrow \left(\bigwedge_k \vec{a}_{ijk} \cdot \begin{bmatrix} \vec{x} \\ \\ \vec{y} \end{bmatrix} \leq b_{ijk} \right)$$

To prove that solving the above formula is in Σ_2^P, we show that with an NP-oracle, we can construct a verification algorithm that runs in polynomial time. Our verifier will take in the certificate composed of $\hat{\Psi}, \hat{\Omega}, \hat{y}$, leaving the subproblem:

$$\neg \exists \vec{x} : \bigwedge_i \bigvee_{j:\hat{\Psi} \models \psi_{ij}} \psi_{ij} \rightarrow \left(\bigwedge_k \vec{a}_{ijk} \cdot \begin{bmatrix} \vec{x} \\ \\ \hat{y} \end{bmatrix} \leq b_{ijk} \right)$$

$$\neg \exists \vec{x} : \bigwedge_i \bigvee_{j:\hat{\Psi} \models \psi_{ij}} \bigwedge_k \vec{a}_{ijk} \cdot \begin{bmatrix} \vec{x} \\ \\ \hat{y} \end{bmatrix} \leq b_{ijk}$$

The unnegated version of this problem is clearly in NP. Given a certificate $\hat{x}$, we can verify whether or not the formula holds in linear time. As a result, with an NP-oracle, we can solve the presented subproblem, meaning that our complement problem is in Σ_2^P and our original problem is thus in Π_2^P. This proves that WC-TCSPU, WC-DTNU, and WC-CDTNU are Π_2^P-complete. □

Dynamic Controllability

Finally, we show that checking dynamic controllability for any temporal network with uncertainty and either disjunctions or conditional constraints is PSPACE-complete.

Theorem A.9. *Checking the dynamic controllability of CSTNUs, TCSPUs, DTNUs, and CDTNUs are PSPACE-complete.*

Proof. We know from Lemma A.2 that DC-TCSPU is PSPACE-hard, meaning that checking the dynamic controllability of DTNUs and CDTNUs must also be PSPACE-hard. Similarly because checking the dynamic controllability of CSTNs is PSPACE-hard [14], DC-CSTNU must also be PSPACE-hard. In order to show that determining dynamic controllability for any of these four networks in PSPACE-complete, we provide an algorithm for checking the dynamic controllability of CDTNUs which requires polynomial space (see Algorithm 14).

Before we explain the details of the algorithm, we need to extend some concepts to describe a partially executed CDTNU, as our algorithm for determining dynamic controllability will recursively act on partially executed networks. We say that events are *scheduled* if they have been assigned a specific value, whether by the scheduler or by nature. We say that a contingent constraint is *active* if its starting event has been scheduled but its ending event has not. Finally, given a group of active contingent constraints, we say that the set of all *realizations* from some time τ is the set of all possible times at which the contingent constraints could end with none of them ending sooner than τ. We now describe the operation of the algorithm before demonstrating that it uses at most polynomial space.

The algorithm works by recursively simulating all possible strategies used by an agent in a dynamically controllable setting. As input, it takes in a list of events whose values have already been scheduled (either by the scheduler or by nature), a list of active contingent constraints, a list of unexecuted events, the CDTNU, and the current time. While there are still executable events that need to be scheduled, the algorithm searches for one that guarantees a valid dynamically controllable strategy.

In the context of dynamic controllability, an agent only has one of two possible

```
**Input:** A list of events with assigned values, $T$;
A list of active contingent constraints, $A$;
A set of yet-to-be-executed events $E$;
The input CDTNU $G$;
The current time, $\tau$
**Output:** Whether the CDTNU is dynamically controllable.
CHECKDC:
if $E.empty()$ then
    for $realization \in A.realizationsFrom(\tau)$ do
        if $!G.isConsistent(T.extend(realization))$ then
            return($false$);
    return($true$);
for $t \in E$ do
    for $\tau' \in [\tau, G.tMax]$ do
        $allSatisfied \leftarrow true$;
        for $realization \in A.realizationsFrom(\tau)$ do
            $earliest \leftarrow realization.earliest()$;
            if $early.time \leq \tau'$ then
                if $!$CHECKDC$(T \cup \{earliest\},$
                $A.nextContingents(earliest),$
                $E,$
                $G,$
                $earliest.time)$ then
                    $allSatisfied \leftarrow false$;
                    break;
            else
                if $!$CHECKDC$(T \cup \{$EVENTASSIGNMENT$(t, \tau')\},$
                $A.nextContingents($EVENTASSIGNMENT$(t, \tau')),$
                $E \setminus t,$
                $G,$
                $\tau')$ then
                    $allSatisfied \leftarrow false$;
                    break;
        if $allSatisfied$ then
            return $true$;
return($false$);
```

Algorithm 14: PSPACE algorithm for checking DC-CDTNU.

actions: they can either unconditionally schedule an action or schedule an action to occur so long as no other contingent event occurs in the interim. We model this behavior by modeling all scheduling actions as interruptible by contingent events. In other words, if a contingent event occurs before an event that is unconditionally scheduled, the algorithm still gives the agent the choice to adapt their strategy. In the case of an unconditionally scheduled action, the agent would just reaffirm their previous choice.

To model all strategies, the algorithm iterates over all possible events that can be scheduled (line 6) and all possible times at which they can be scheduled (line 7). If at least one scheduling of an event given the input parameters is valid, then the CDTNU is dynamically controllable. When there are no active contingent constraints that might be scheduled before the event that is chosen to be schedule, the algorithm will recurse on that assignment to get an answer (line 20-26). In the case that there are contingent constraints that may occur earlier, the algorithm must respond to them in turn (lines 9-18). If all possible realizations of contingent constraint values still guarantee that the CDTNU is consistent, then the system is dynamically controllable.

Now, we show that the algorithm uses at most polynomial space. If there are no more events to schedule, then the algorithm remains in lines 1-5 of the algorithm, which check consistency over all possible realizations of the remaining contingent constraints. Checking consistency is a polynomial time operation, as it just requires iterating through each constraint and verifying that it is satisfied. While there are exponentially many such realizations, the algorithm does not have to store each particular realization; instead it merely has to remember the current realization and know how to increment to the next one. As a result, handling all realizations also takes polynomial space.

In the event that there are executable events to schedule, the algorithm operates over lines 6-29. Iterating over each event at line 6 takes polynomial space, and when iterating over all τ' at line 7, there exist an exponentially large but finite number of values to choose from, but it only takes polynomially many bits to represent such a choice. At line 9, the algorithm handles realizations in the same way it did at line 2, meaning it only needs polynomial space, and then the space needed for the remaining recursive calls. If we look at the number of possible stack frames, we see that every time CHECKDC is called, the algorithm adds a new event to T, correspondingly removing it from either A (line 13) or E (line 22). The set E never grows, and because every contingent constraint's ending event is unique, the algorithm will never add the same event to A twice. This means that after at most $|X_e \cup X_b|$ recursive calls, the algorithm will reach a state where E is empty, and the recursive calls will terminate. This preserves the guarantee that the algorithm use polynomial space, meaning that Algorithm 14 decides DC-CDTNU and is in PSPACE. Thus, deciding dynamic controllability for CSTNUs, TCSPUs, DTNUs, and CDTNUs are all PSPACE-complete. □

A.2 Multi-Agent Disjunctive Temporal Networks

To this point, we have focused on single-agent temporal network formalisms that use particular types of constraints to emulate multi-agent behavior. While it is reasonable to do so, these approaches occasionally have their shortcomings. Such models either assume that all agents can be jointly controlled, which is too strong of an assumption in practice, or more conservatively, that the actions of other agents are themselves modeled as temporally uncertain processes and cannot be strategically reasoned about. While such approaches provide some correctness guarantees, they require agents to act in a way that is robust to all possible actions that other agents might take rather than reasoning over joint strategies that permit coordination without absolute control of all other agents.

In this section, we introduce multi-agent temporal networks with disjunctions and uncertainty and provide completeness results for their complexity. We demonstrate that the addition of temporal uncertainty to multi-agent disjunctive temporal network problems raises the complexity to PSPACE-completeness in the case of Partially Observable Disjunctive Temporal Networks with Uncertainty (PODTNUs) and to NEXP-completeness in the case of Multi-agent Disjunctive Temporal Networks with Uncertainty (MaDTNUs). The contributions of this section are summarized in Figure A-5. These results represent the first completeness results for multi-agent temporal networks. While these results indicate that using multi-agent networks directly is likely too computationally expensive in practice, these results are instrumental in

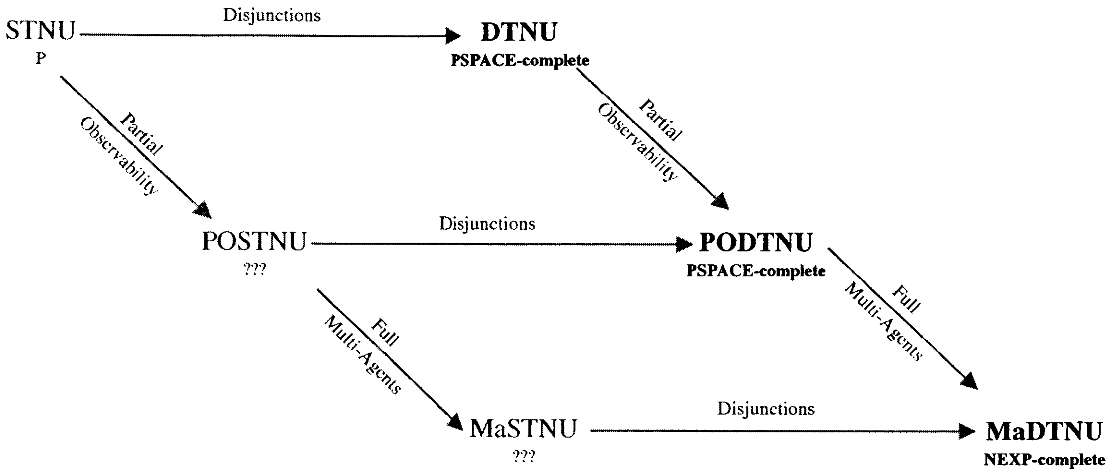


Figure A-5: A taxonomic organization of temporal networks considered in the second section of this appendix, how they relate to one another, and the complexity classes to which their decision problems belong. Results in bold represent novel results provided in this thesis.

shaping how we choose to tackle the multi-agent modeling problem going forward.

A.2.1 Motivation for Multi-Agent Extensions

Before we explore these new network variants, it is worth considering whether these new networks give us additional expressive power to model interesting behavior. Consider an example that involves a series of humans and robots in a warehouse working collaboratively to fill orders. Two dexterous (potentially heterogeneous) picker robots are tasked with retrieving individual items from bins across the warehouse and a third delivery robot is tasked with taking bins of objects that compose orders to a group of human packers who will inspect the items for defects before placing them in a box with appropriate packing materials.

We have two new orders come in each with two different objects, and the two picker robots are tasked with retrieving one item from each order as matches their skill sets. Retrieving a single item takes 20-30 minutes, and each picker locally has the flexibility to choose the order in which it undertakes tasks. Once a bin of objects is assembled, it takes 15 minutes for the delivery robot to bring it to the human packer. It takes a human packer 15 minutes to inspect and assemble a package once the items are delivered.

In this problem, warehouses must maintain a high level of throughput, imposing temporal deadlines, and traffic within the warehouse and the variable difficulties of grasping tasks implies that there is temporal variability in the execution of actions. It's clear that there needs to exist some capability of representing choices (i.e. choose which object to grab first) and disjunctive constraints give us a powerful way to do that (i.e. we require that either the pickup of object 1 ends before pickup of object 2 starts or vice versa). The remaining question is whether it suffices to use a series of single-agent simplifications, such as DTNs or DTNUs, to model this problem.

Unfortunately, the answer is no. A single-agent projection of our problem allows a robot to freely choose the ordering of their own actions but the actions of others are then abstracted to entirely stochastic, uncorrelated and uncontrollable actions. If the two packages must be fully prepared within 90 minutes, we know the task is impossible. The first picker robot cannot know a priori which object the second one will grab first and so the two may choose objects from different orders. If it took both robots the full 30 minutes to grab the orders, then the delivery robot could only deliver the packages after an hour; the packer would get the deliveries after 75 minutes and would need at least 30 to finish with both.

In contrast, modeling the system using multi-agent temporal networks, as is done with POSTNUs and MaSTNUs, would allow us to correctly determine that all constraints can be satisfied. Though the pickers may not know what the other is doing in real-time, they can adopt contingency and coordination strategies that eliminate much of the network's cross-agent uncertainty. PODTNUs and MaDTNUs allow for the encoding of multi-agent coordination strategies that is are common across many other game-playing formalisms.

A.2.2 Multi-agent Disjunctive Definitions

Extending DTNUs to include partial observability yields Partially Observable Disjunctive Temporal Networks with Uncertainty (PODTNUs). PODTNUs make a distinction between contingent events that are observable and unobservable, allowing the modeler to chain together contingent constraints to model shared causes of stochasticity.

Definition A.8. PODTNU

A PODTNU is a 5-tuple $\langle X_e, X_c, X_u, R_r, R_c \rangle$ where:

- X_e is the set of executable events
- X_c is the set of observable contingent events
- X_u is the set of unobservable contingent events
- R_r is the set of full disjunctive temporal constraints indexed by k, called requirement constraints, of the form $\bigvee\limits_k (l_{r,k} \leq x_{r,k} - y_{r,k} \leq u_{r,k})$, where $x_{r,k}, y_{r,k} \in X_e \cup X_c \cup X_u$ and $l_{r,k}, u_{r,k} \in \mathbb{R}$
- R_c is the set of simple disjunctive contingent constraints indexed by k, of the form $x_r - y_r \in \bigcup\limits_k [l_{r,k}, u_{r,k}]$, where $y_r \in X_e \cup X_c \cup X_u$, $x_r \in X_c \cup X_u$, and $l_{r,k}, u_{r,k} \in \mathbb{R}$

PODTNUs are powerful because they allow us to model shared dependencies between different events, but they still make the assumption that external agents act without regard to the ego agent's goals and constraints. To truly take advantage of multi-agent coordination that we see in multi-agent interaction, we should not assume randomness from an agent's collaborators. Instead, we should recognize that we can partially coordinate joint approaches to guarantee constraint satisfaction. With POSTNUs, we accomplish this by extending our model to Multi-agent Simple Temporal Networks with Uncertainty (MaSTNUs) [15], and for PODTNUs, we can similarly extend our formalism to that of Multi-agent Disjunctive Temporal Networks with Uncertainty (MaDTNUs).

Definition A.9. MaDTNU

An MaDTNU is a 5-tuple $\langle A, X_e, X_c, R_r, R_c\rangle$ where:

- A is a (non-empty) set of agents
- X_e is the set of executable events
- X_c is the set of contingent events
- R_r is the set of full disjunctive temporal constraints indexed by k, called requirement constraints, of the form $\bigvee\limits_k (l_{r,k} \leq x_{r,k} - y_{r,k} \leq u_{r,k})$, where $x_{r,k}, y_{r,k} \in X_e \cup X_c$ and $l_{r,k}, u_{r,k} \in \mathbb{R}$
- R_c is the set of simple disjunctive contingent constraints indexed by k, of the form $x_r - y_r \in \bigcup\limits_k [l_{r,k}, u_{r,k}]$, where $y_r \in X_e \cup X_c$, $x_r \in X_c$, and $l_{r,k}, u_{r,k} \in \mathbb{R}$

The set of events $X = X_e \cup X_c$ is partitioned across all agents in A, such that each event is assigned to exactly one agent. We generally carve out an exemption for a single anchor event Z, visible to all agents, that represents the start of all execution. MaDTNUs require us to specify the observability of each event during execution, as the observation of events by specific agents can significantly aid in the eventual success of the scheduling process. With MaDTNUs, we assume that each event, whether executable or contingent, can only be observed by the agent the event is assigned to. In order to make an event observable to another agent, it suffices to add a new contingent constraint from the original event to a new contingent event with zero duration that is observable by the second agent. The second agent should be able to infer the timing of the original event from the contingent event they can observe.

When we consider the feasibility of temporal networks with uncertainty, like PODTNUs and MaDTNUs, we cannot just validate a statically constructed schedule. The actual outputted schedule depends heavily on the conditions under which we observe the temporally uncertain events. As such, we often consider a temporal network's *controllability* in determining whether or not it is possible to construct a schedule. We traditionally care about the *strong, dynamic, or weak controllability* of a network, concepts that are adapted from the evaluation of STNUs [58].

Dynamic controllability tends to be the most interesting of the three forms of controllability as it provides the agents the power to react to the actual situations that manifest themselves during execution. As such, the remainder of this appendix will focus on understanding the complexity of determining the dynamic controllability of these networks; we will call these problems DC-PODTNU and DC-MaDTNU for short.

A.2.3 PODTNU Controllability

In the case of partially observable temporal networks, determining strong and weak controllability reduces to computing the same type of controllability over a fully observable version of the same network. This follows naturally from the definitional differences between the two different types of networks. Strong controllability assesses whether a schedule can be obstructed in absence of any observations, in practice making all contingent events unobservable, whereas weak controllability assesses whether a schedule can always be constructed when given perfect foresight, adding a condition even stronger than making all contingent events observable.

The same reasoning cannot be applied to dynamic controllability in PODTNUs. When dynamically executing a PODTNU, certain values remain hidden while others are readily exposed upon execution. What is quite noteworthy, however, is that the revealed value of certain observable contingent events may reveal information about other unobservable ones.

This makes the question of determining DC-PODTNU more involved than that of determining dynamic controllability for ordinary DTNUs. In this section, we will show that despite this difference, DC-PODTNU complexity matches that of dynamic controllability for DTNUs, which is PSPACE-complete [8]. In order to prove that DC-PODTNU is PSPACE-complete, we must show that it is both PSPACE-hard and that it is solvable in PSPACE. We know that PODTNUs generalize DTNUs, as a DTNU is a PODTNU without unobservable contingent events. Because we know checking the dynamic controllability of a DTNU is PSPACE-hard [8], we thus know that DC-PODTNU is PSPACE-hard. What remains is to show that DC-PODTNU is solvable in PSPACE.

Theorem A.10. *DC-PODTNU $\in$ PSPACE.*

Proof. In order to demonstrate this, we will provide an algorithm that checks the dynamic controllability of a PODTNU (Algorithm 15) and show that it runs in PSPACE. Our algorithm is adapted heavily from the algorithm for checking dynamic controllability of Conditional Disjunctive Temporal Networks with Uncertainty [8] and as such relies on the fact that at most a polynomial number of bits are being used to represent event values. Our algorithm, however, makes no assumptions about how specifically numbers are represented and only relies on the ability of a computer to iterate through all representable numbers.

Our algorithm operates on our original PODTNU P and makes use of a DTNU D derived from it. D is constructed by removing all unobservable events and modifying the related set of temporal constraints. We start with requirement constraints that involve unobservable contingent events. For a given unobservable contingent event B, we eliminate all disjuncts of requirement constraints that involve B. Note that if a fully disjunctive temporal constraint involves B in every disjunct, then the entire constraint is eliminated. Next, we consider all contingent constraints that involve B. If B is involved in more than one contingent constraint then it must be in the form $A \overset{R_1}{\Longrightarrow} B \overset{R_2}{\Longrightarrow} C$ since each contingent constraint has a unique endpoint. We eliminate B in D by replacing each such chain of contingent constraints with a new single contingent constraint $A \xRightarrow{R_1+R_2} C$, where the new constraint bounds are given using standard interval arithmetic [52]. After recursively applying this procedure, we have eliminated all constraints involving unobservable contingent events and thus can safely remove those events to turn our PODTNU into a DTNU. We then feed in the original PODTNU P and the derived DTNU D into our algorithm, CHECKPODC.

CHECKPODC works by recursively enumerating all possible strategies for assignments to executable events and accurately simulating all possible observable events

```
Input: A list of events with assigned values, $T$;
A list of active contingent constraints, $A$;
A set of yet-to-be-executed events $E$;
The input PODTNU $P$;
$P$'s projection to a DTNU, $D$;
The current time, $\tau$
Output: Whether the PODTNU is dynamically controllable.
CheckPODC:
if $E.empty()$ then
    for $realization \in D.realizationsFrom(A, \tau)$ do
        $T' \leftarrow T.extend(realization)$;
        for $unobsRealiz \in P.unobsRealizFrom(T')$ do
            if $contingentMismatch(unobsRealiz, T')$ then
                continue;
            if $!P.isConsistent(T')$ then
                return($false$);
    return($true$);
for $t \in E$ do
    for $\tau' \in [\tau, P.tMax]$ do
        $allSatisfied \leftarrow true$;
        for $realization \in D.realizationsFrom(A, \tau)$ do
            $earliest \leftarrow realization.earliest()$;
            if $earliest.time \leq \tau'$ then
                if !CheckPODC($T \cup \{earliest\}$,
                $A.nextContingents(earliest)$,
                $E, P, D, earliest.time$) then
                    $allSatisfied \leftarrow false$;
                    break;
            else
                if !CheckPODC($T \cup \{$EventAssignment$(t, \tau')\}$,
                $A.nextContingents($EventAssignment$(t, \tau'))$,
                $E \setminus t, P, D, \tau'$) then
                    $allSatisfied \leftarrow false$;
                    break;
        if $allSatisfied$ then
            return $true$;
return($false$);
```

Algorithm 15: PSPACE algorithm for checking DC-PODTNU.

of the inputted DTNU. At any given call to CheckPODC, some event values have been fixed, which we denote by T, and some contingent constraints have their starting event executed but their ending event still unexecuted, which we denote by A.

The algorithm starts by picking an unexecuted event (line 10) and a time at which to execute it (line 11). It then looks over all possible values of the ending contingent constraints of A (line 13) and checks whether it comes before our stated execution time. If it does, the algorithm checks whether our strategy still holds after observing the ending contingent event (lines 16-20), and if not, it executes our chosen event and continue onwards (lines 22-26). The algorithm uses $allSatisfied$ to keep track of whether for any particular choice of event to execute, every possible scenario still guarantees success.

It is important to note that at this point, the algorithm does not consider the effect of requirement constraints and that the contingent constraints that are informing our search come from D and not P. It is possible then that our choice of values for contingent events differs from what is possible in P. For example if we originally had contingent constraints $A \overset{[0,10]}{\Longrightarrow} B$, $B \overset{[0,1]}{\Longrightarrow} C$, and $B \overset{[0,1]}{\Longrightarrow} C'$ in P, where only B was unobservable, our transformation to D would give us edges $A \overset{[0,11]}{\Longrightarrow} C$, and $A \overset{[0,11]}{\Longrightarrow} C'$. This would suggest that we might be able to have $A = 0, C = 0$, and $C' = 11$, but this is inconsistent with P. We will remedy this problem shortly.

Eventually, the algorithm reaches a point where E is empty because in each recursive step, it either assigns a contingent event (line 18) or assigns an executable event (line 22). When E is empty, it then checks for feasibility. At this point, the algorithm considers the constraints of P. It first assigns any unassigned observable events (line 2) and then iterates through all possible values for the unobservable contingent values that were unassigned during execution (line 4). If it discovers a scenario where the choice of observable events does not match the contingent constraints of the POSTNU, the algorithm skips that particular choice (lines 5-6). If it is a valid configuration, the algorithm then checks whether the POSTNU constraints are satisfied. If the POSTNU constraints are satisfied across all possible valid choices of unobservable contingent events, the algorithm returns true, and if not, it returns false. Thus, our procedure returns the correct answer because it considers execution strategies operating over a superset of all possible situations of POSTNU P and returns true if there exists an execution strategy that satisfies all valid realizations of uncertain values.

What remains is to show that the algorithm uses at most polynomial space. If n is the number of events in our graph, there are at most n recursive calls at any one given time since each call assigns a new value to a variable. The algorithm iterates over each event (line 10), each potential time (line 11), and each realization of contingent values (line 13). While there are exponentially many values each of these can take on, writing them down requires only polynomially many bits. Finally checking for valid contingent event values when adding observable values and checking overall PODTNU consistency takes linear time (and thus at most linear space) and can be done simply by iterating through each constraint and checking correctness. Thus, the procedure determining dynamic controllability requires at most polynomial space.

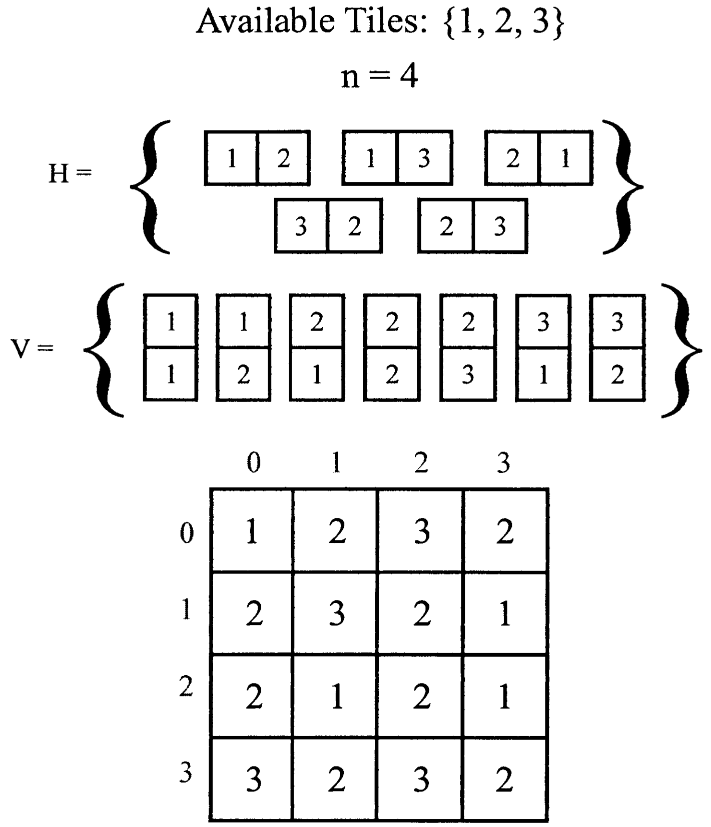
□

Because DC-PODTNU is PSPACE-hard and can be determined with polynomial space, we know that DC-PODTNU is PSPACE-complete.

A.2.4 MaDTNU Controllability

When we expand multi-agent disjunctive reasoning to consider agents that can coordinate their strategies, the task of determining controllability becomes much more difficult.

To show that DC-MaDTNU is NEXP-hard, we first introduce the TILING problem which is itself NEXP-complete [2, 43]. The TILING problem asks whether it is possible to number a $n \times n$ board according to the following rules (see Figure A-6 for reference). Each tile on the board can be filled in with a number from 1 to m, and there are a set of horizontal and vertical pairwise rules, respectively H and V, that indicate how numbers can adjoin each other on the board. We say that $f : \{0, 1, ..., n-1\} \times \{0, 1, ..., n-1\} \rightarrow \{1, 2, ..., m\}$ is a tiling function mapping from each of the row and column indices to the number on that tile. In order to ensure that the horizontal and vertical pairwise rules are respected, we require that $\forall i \in \{0, 1, ..., n-1\}, \forall j \in \{0, 1, ..., n-2\} : \langle f(i,j), f(i,j+1)\rangle \in H$ and



	0	1	2	3
0	1	2	3	2
1	2	3	2	1
2	2	1	2	1
3	3	2	3	2

Figure A-6: Example TILING problem with accompanying solution.

$\langle f(j,i), f(j+1,i)\rangle \in V$. We say that a solution for the TILING problem exists if there exists an f satisfying the pairwise rules with $f(0,0) = 1$. Note that since it takes $u = \lceil \log n \rceil$ bits to represent n, enumerating a tiling function to serve as a certificate may take exponential time.

In order to prove hardness, we simply show that TILING is reducible to DC-MaDTNU.

Lemma A.11. *DC-MaDTNU is NEXP-hard.*

Proof. To show that DC-MaDTNU is NEXP-hard, we demonstrate how to construct an MaDTNU that is dynamically controllable if and only if a corresponding TILING problem has a valid solution.

Our strategy for the reduction is to give each of two agents a location on the TILING grid and have them report back a tile value to put at that spot without knowing which location was provided to the other agent. As such, our MaDTNU construction must first simulate the hidden random TILING grid location selection and must also enforce adjacency rules if the two agents are given locations that adjoin one another.

We construct our two-agent MaDTNU as follows (see Figure A-7). The MaDTNU has a single event Z that is observable by all (which for convenience we will assume is always assigned at time $t = 0$), and all other events will be visible to exactly one of the agents. Each agent will have $2u$ contingent constraints that the other cannot see, and we say that each contingent constraint is made up of events $A_{i,p} \xRightarrow{[0,0]\vee[2^i,2^i]} C_{i,p}$, where i is the index of the contingent constraint (from 0 to $2u-1$) and p represents the player it corresponds to. We add a requirement constraint $Z \xrightarrow{[0,0]} A_{0,p}$ for each agent as well as requirement constraints $C_{i,p} \xrightarrow{[0,0]} A_{i+1,p}$ for each i and p. We finally add new events X_1, X_2 with requirement constraints enforcing $C_{2u-1,p} \xrightarrow{\{1,2,\ldots,m\}} X_p$.

With the given structure, we have a way to implicitly select a spot on the original TILING grid for each agent. The first u contingent constraints represent the selection of the row index and the second u contingent constraints represent the column index. Specifically, we represent row index r and column index c by assigning contingent constraints in a way that ensures $C_{2u-1,p}$ occurs at time $r + c \cdot 2^u$. It is worth noting that in instances where n is not a power of two, there will be some assignments of contingent constraints that do not correspond to valid positions on the TILING grid; we will handle these situations when we consider how to enforce adjacency rules.

Before we add any of the tiling adjacency constraints, it is clear that our MaDTNU is dynamically controllable. Each $A_{i+1,p}$ is assigned immediately when $C_{i,p}$ is assigned and any valid tile value can be picked to satisfy the requirement constraints associated with X_1 and X_2. We now add constraints to enforce that adjacent tiles respect the pairwise tiling rules and show how those new constraints are sufficient to complete the reduction.

First, we consider how to handle contingent constraint values that are not valid tiling locations. In these cases, we should assume that any tiling choices are valid and ensure that all constraints are satisfied in those instances. To accommodate this for each of the constraints ψ we introduce, we amend the constraint to instead be

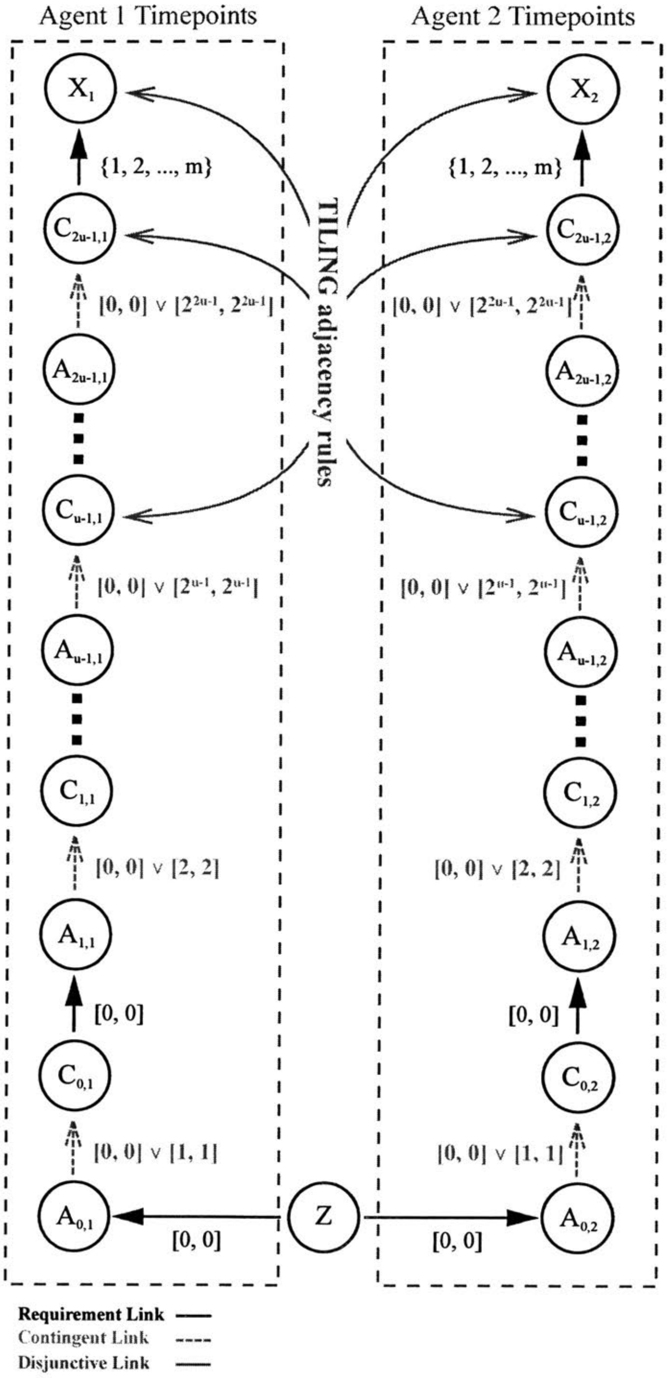


Figure A-7: The two-agent MaDTNU produced by a reduction from an input TILING problem. There are $O(\log n)$ events in total and $O(|H| + |V| + \log n)$ constraints in total, each of which are $O(|H| + |V|)$ in size.

$\phi_{oob} \vee \psi$, where:

$$\phi_{oob} = (C_{u-1,1} - Z \geq n) \vee (C_{u-1,2} - Z \geq n) \vee$$

$$(C_{2u-1,1} - C_{u-1,1} \geq 2^u \cdot n) \vee (C_{2u-1,2} - C_{u-1,2} \geq 2^u \cdot n)$$

In the instance that the contingent constraints are realized such that the corresponding tiling location is outside the established bounds, all constraints vacuously hold, and the network is dynamically controllable.

Now, we move on to encoding the pairwise rules. For convenience, we will use the shorthand T_p to represent $X_p - C_{2u-1,p}$. We write the horizontal rules as follows, starting with the instance where agent 1 must pick a tile to the left that of agent 2:

$$\bar{\phi}_{oob} \wedge (C_{2u-1,2} - C_{2u-1,1} = 2^u) \implies \langle T_1, T_2 \rangle \in H$$

To simplify our exposition, we will split this constraint into several constraints that vary based on agent 1's choice for T_1. In other words, for each $j \in \{1, 2, ..., m\}$, we now consider the constraint:

$$\bar{\phi}_{oob} \wedge (C_{2u-1,2} - C_{2u-1,1} = 2^u) \wedge (T_1 = j)$$

$$\implies \langle j, T_2 \rangle \in H$$

We can rewrite our equation to eliminate the implication, and since all of our values are guaranteed to be integers by construction, we can rewrite any statements involving $\neq$ with inequalities on both sides. Finally, we know that $\langle j, T_2 \rangle \in H$ is equivalent to $\bigvee_{l \in \{1,2,...,m\}:\langle j,l \rangle \in H} T_2 = l$, and substituting those values in we get:

$$\phi_{oob} \vee (C_{2u-1,2} - C_{2u-1,1} \geq 2^u + 1) \vee$$

$$(C_{2u-1,2} - C_{2u-1,1} \leq 2^u - 1) \vee (T_1 \geq j + 1)$$

$$\vee (T_1 \leq j - 1) \vee \bigvee_{l \in \{1,2,...,m\}:\langle j,l \rangle \in H} T_2 = l$$

It is worth noting that this approach adds $O(|H|)$ constraints each of which is $O(|H|)$ in size, meaning that our reduction still requires at most polynomial time. For completeness, we also must consider the case where agent 1 picks a tile to the right of agent 2. This requires switching the roles of agents 1 and 2 in the rules:

$$\bar{\phi}_{oob} \wedge (C_{2u-1,1} - C_{2u-1,2} = 2^u) \implies \langle T_2, T_1 \rangle \in H$$

and again, this can be expanded for each $j \in \{1, 2, ..., m\}$ into a simple disjunctive constraint of the form:

$$\phi_{oob} \vee (C_{2u-1,1} - C_{2u-1,2} \geq 2^u + 1) \vee$$

$$(C_{2u-1,1} - C_{2u-1,2} \leq 2^u - 1) \vee (T_2 \geq j + 1)$$
$$\vee (T_2 \leq j - 1) \vee \bigvee_{l \in \{1,2,...,m\}:\langle j,l \rangle \in H} T_1 = l$$

We take a similar approach for satisfying the vertical rules, seeing if the final contingent constraint values of the two agents differ by exactly one, but in this case, we now need to make sure that the two values are still in the same column instead of being wrapped around to a new one. To accommodate this, we now have to additionally verify that the larger of the two tile indices is not in the 0$^{\text{th}}$ row. This yields constraints of the form:

$$\bar{\phi}_{oob} \wedge (C_{2u-1,2} - C_{2u-1,1} = 1) \wedge (C_{u-1,2} \neq 0)$$

$$\implies \langle T_1, T_2 \rangle \in V$$

and:

$$\bar{\phi}_{oob} \wedge (C_{2u-1,1} - C_{2u-1,2} = 1) \wedge (C_{u-1,1} \neq 0)$$

$$\implies \langle T_2, T_1 \rangle \in V$$

By applying the same approach as with the horizontal rules, we again create distinct constraints for each $j \in \{1, 2, ..., m\}$ and can rewrite our rules to get simple disjunctive constraints:

$$\phi_{oob} \vee (C_{2u-1,2} - C_{2u-1,1} \geq 2) \vee$$

$$(C_{2u-1,1} - C_{2u-1,2} \leq 0) \vee (C_{u-1,2} = 0)$$

$$\vee (T_1 \geq j + 1) \vee (T_1 \leq j - 1)$$

$$\vee \bigvee_{l \in \{1,2,\ldots,m\}:\langle j,l \rangle \in V} T_2 = l$$

and:

$$\phi_{oob} \vee (C_{2u-1,1} - C_{2u-1,2} \geq 2) \vee$$

$$(C_{2u-1,1} - C_{2u-1,2} \leq 0) \vee (C_{u-1,1} = 0)$$

$$\vee (T_2 \geq j + 1) \vee (T_2 \leq j - 1)$$

$$\vee \bigvee_{l \in \{1,2,\ldots,m\}:\langle j,l \rangle \in V} T_1 = l$$

Finally, for the sake of simplicity, we will also require that if the two agents receive the same value from their contingent constraints, then they must report the same T_p. This adds the constraint:

$$\bar{\phi}_{oob} \vee (C_{2u-1,1} = C_{2u-1,1}) \implies T_1 = T_2$$

which can be rewritten as:

$$\phi_{oob} \vee (C_{2u-1,1} \geq C_{2u-1,1} + 1)$$

$$\vee (C_{2u-1,1} \leq C_{2u-1,1} - 1) \vee (X_1 - X_2 = 0)$$

It is clear by construction that if a solutions exists for a TILING grid then the MaDTNU we constructed is dynamically controllable. An acceptable strategy is for both agents to pre-compute a shared valid tiling and to use that tiling to pick X_1, X_2 based on the values of the contingent constraints each can observe. What remains is to demonstrate that knowing that our constructed MaDTNU is dynamically controllable implies that the corresponding TILING problem has a valid solution.

Our decision to add a constraint requiring that both agents report the same value when queried for the same tile simplifies our analysis, as it requires the two agents to have the same, deterministic strategy. Thus, our problem reduces to being able to prove that a solution to the TILING problem exists if the agents have a strategy that renders the network controllable.

We start with the observation that each agent really has only one decision point, namely when to schedule X_p. Each of the intermediate $A_{i,p}$s are scheduled immediately after the corresponding preceding contingent events, so there are no real decisions to be made in scheduling $A_{i,p}$. There are 2^{2u} possible values that event $C_{2u-1,p}$ can be assigned to, but thanks to ϕ_{oob}, we only care about n^2 of them. If we take the agent's strategy for those n^2 time points that correspond to a row-column indexing into the original grid, we can immediately translate that into a valid tiling. We prove this by contradiction.

Assume momentarily that translating the first agent's strategy does not yield a viable tiling. If the agent's strategy were stochastic, we could take any possible grounded strategy and use that as our main one, as controllability implies that all constraints are satisfied across all possible agent strategy realizations. This means that for the tiling to be invalid, there must be a pair of tiles that adjoin one another that violate the original tiling rules. Let's call these indices i and j and the corresponding tiles assigned by agent 1 for these indices T_i and T_j.

We know that in order for the system to be controllable, agent 2's strategies for i and j must also be to assign T_i and T_j because of the rule that requires reporting the same values when the contingent constraint values are all the same. Because our original MaDTNU was known to be dynamically controllable, this means that for all possible uncertain values all constraints are satisfied, including when agent 1 must assign a value for index i and when agent 2 must assign a value for index j. In this situation, they must produce values T_i and T_j, respectively, but because indexes i and j adjoin one another and all constraints are known to be satisfied, then T_i and T_j must satisfy the appropriate adjacency relation. This means that the tiling derived from the strategy cannot have a pair of indices that adjoin one another and violate a rule, concluding the proof that TILING is reducible to DC-MaDTNU and that

DC-MaDTNU is NEXP-hard.

□

Our reduction demonstrates that DC-MaDTNU is NEXP-hard. Now, we show that DC-MaDTNU can be solved by a non-deterministic Turing machine in exponential time.

Lemma A.12. *DC-MaDTNU $\in$ NEXP.*

Proof. To show that DC-MaDTNU $\in$ NEXP, we will generate strategies for each agent and show that the joint execution of these strategies guarantees success.

First, we need to generate strategies for each agent. We know that it is possible to describe a dynamic execution strategy for an agent in a DTNU that can be efficiently executed [17]. Even though these strategies when enumerated can be exponential in the size of the input, this will suffice for our purposes. Note that an important part of this approach is that we assume that it takes a fixed number of bits (though possibly polynomially many) to represent individual numbers but is agnostic as to how specifically numbers are represented.

In order to construct a strategy for the overall MaDTNU, we will start by guessing a strategy for each agent with respect to their locally projected events. We define the locally projected events of an agent in an MaDTNU as the collection of events that are directly observable by that agent. Whereas events in the MaDTNU were subdivided into events assigned by the ego agent, events assigned by nature, and events assigned by each of the other agents, the locally projected events will only be subdivided into events assigned by the ego agent and events assigned by others.

Given a set of locally projected events, we guess a DTNU strategy non-deterministically over that set of events. Note that we are guessing this strategy without explicit knowledge of any dependencies between events or decisions strategically made by other agents; at this moment, we are leaning on non-determinism to generate local strategies for each agent that are together globally consistent.

The strategies we generated can each be exponentially large, but, importantly, it takes at most exponential time to guess an exponentially large string. Through the

```
Input: An MaDTNU $G$
Output: Whether $G$ is dynamically controllable.
Initialization:
strategies ← guessed local strategies for each agent;
DC-MaDTNU:
for $\omega \in G.uncertainRealizations()$ do
    $assigned \leftarrow \emptyset$;
    $time \leftarrow 0$;
    while $len(assigned) < G.numEvents()$ do
        $possibleActions \leftarrow \{s.nextAction(assigned, time) \mid s \in strategies\}$;
        $nextAction \leftarrow possibleActions.earliest()$;
        $possibleConts \leftarrow \{\langle y, t\rangle \mid y \notin assigned \wedge \exists x \in assigned : \langle x, y, t\rangle \in$
          $G.contingents()\}$;
        $nextCont \leftarrow possibleConts.earliest()$;
        if $nextAction.before(nextCont)$ then
            $assigned.add(\langle nextAction.event(), nextAction.time()\rangle)$;
            $time \leftarrow nextAction.time()$;
        else
            $assigned.add(\langle nextCont.event(), nextCont.time()\rangle)$;
            $time \leftarrow nextCont.time()$;
    if $G.constraintsViolated(assigned)$ then
        return($false$);
return($true$);
```

Algorithm 16: NEXP algorithm for checking DC-MaDTNU.

strategy generation step, we are still well within our established time bounds.

Now, we must validate that the strategies we guessed ensure dynamic controllability. Or that for any possible realization of the uncertain duration of constraints, we can still guaranteeably satisfy all constraints. We do so by brute force iteration (see Algorithm 16).

Our brute force enumeration relies on the fact that the bits required to encode any particular uncertain state are polynomial in the problem input size. In other words, writing down a realization of contingent constraint values takes at most polynomial space even though there are exponentially many such realizations.

For any given eventual realization of contingent constraint durations, a fixed strategy will yield a deterministic output. Our algorithm gives a straightforward simulation operating on behalf of each individual agent. We build out our simulation by iteratively grounding the values of individual events based on agent strategies and the different realized durations of contingent constraints; these values are stored in the *assigned* variable (line 3).

Each agent's strategy only allows them to observe a subset of events and make decisions off of those events, and those decisions reduce to unconditionally executing an event at a certain point in time or conditionally waiting to observe an uncontrolled event before making a decision. In the event that an agent's action is to conditionally wait for another event, we instead record the agent that action would take if that uncontrolled event took on its latest allowable value; these actions are chosen at line 6 from the nondeterministically guessed agent strategy.

While some uncontrollable actions are chosen by other agents, some are controlled by nature through contingent constraints. Though the durations of these contingent constraints are determined when we grounded them (line 2), their values are not yet visible to the agents and so must be learned iteratively. We can imagine that nature behaves like a non-cooperative (or for the sake of controllability checking, even adversarial) agent in the way that it picks its events, and so similarly consider the next contingent values to be realized (line 8). We update the *assigned* variable one event at a time, selecting the earliest values from the derived agent actions and contingent constraint values (lines 7, 9, 10) to ensure that strategies can be adjusted based on new information.

Now we show that the process as a whole takes at most exponential time on a non-deterministic Turing machine.

We have already demonstrated that it takes exponential time to generate the strategies at line 1 of Algorithm 16, and since there are exponentially many realizations of contingent constraint uncertainties, we turn our focus to the runtime of the body of the for loop at line 2. The while loop goes through $O(n)$ iterations in total since *assigned* grows by one after each iteration (lines 11 and 14). The generation of *possibleConts* at line 8 takes at most $O(n)$ time since there are at most n contingent constraints, but the most expensive part of the process is the generation of *possibleActions* at line 6. Since each strategy is exponentially large, and it takes time linear in strategy size to determine what action to take next, the strategy generation at line 6 takes exponential time. However, that this exponential time operation happens an exponential number of times still guarantees that the overall runtime of the algorithm is exponential. Thus, Algorithm 16 runs in NEXP time, and DC-MaDTNU $\in$ NEXP.

□

By Lemma A.11, we know that DC-MaDTNU is NEXP-hard, and by Lemma A.12, we know that DC-MaDTNU $\in$ NEXP. Thus, DC-MaDTNU is NEXP-complete.

A.3 Discussion

Our work provides novel complexity results that are much tighter than existing bounds and require at most polynomial space for strong, weak, and dynamic controllability of several distinct types of temporal networks; these results summarized in Figures A-1 and A-5. Beyond the contribution of the relevant proofs, the value of these results is that it gives modelers insight into which types of features have a significant impact on the runtime complexity of a problem. Many of these insights are not immediately obvious, and in the remainder of this section we discuss a few of them.

First we consider CSTNUs. CSTNUs are a generalization of CSTNs and STNUs and share much in common with their predecessors. In particular, strong controllability of CSTNUs, being in P, can be computed quite efficiently. Our proof for Theorem A.4 actually proves a stronger result that a CSTNU is strongly controllable if and only if the corresponding STNU derived by making all constraints unconditional is strongly controllable. This implies that strong controllability of CSTNUs can be computed in $O(mn)$ time, which is as fast as it takes to compute the feasibility of a simple STN. When we turn to weak and dynamic controllability, we see that checking the controllability of a CSTNU is in the same class as checking controllability of a CSTN. From the perspective of the modeler, this implies that there is a surprisingly low cost to adding uncertainty to a temporal constraint model that already uses conditional constraints.

While CSTNU controllability checking matches the complexity of CSTN controllability checking, it only matches the controllability checking complexity of strong and weak controllability for STNUs. In fact, dynamic controllability checking across all types of single-agent networks, with the exception of STNUs, is PSPACE-hard. In scheduling problems, modelers must often make the trade-off between using strong controllability, which is often easier to compute, and dynamic controllability, which gives more flexibility during execution but is more expensive. In instances where dynamic controllability is deemed necessary, there is a significant advantage to relaxing the underlying temporal model, eliminating conditional and disjunctive constraints, to use an STNU. It is still quite surprising that despite the fact that STNU dynamic controllability can be determined in polynomial time, every other modification makes computing dynamic controllability at least PSPACE-hard.

The complexity of controllability for single-agent disjunctive temporal networks also yields interesting results. The two single-agent temporal network models that use disjunctions without temporal uncertainty are TCSPs and DTNs; TCSPs have simple disjunctions, only requiring disjunctions over a single constraint, while DTNs have full disjunctions, allowing disjunctions to span multiple constraints. Since determining feasibility for both network structures is NP-complete, intuition would suggest that after adding uncertainty the complexity of checking controllability for TCSPUs and DTNUs would also be the same. While this is the case for weak and dynamic controllability, we do see a difference in strong controllability, meaning that strong controllability is easier to compute in TCSPUs than it is in DTNUs, assuming NP $\neq \Sigma_2^P$, implying there is a meaningful difference between the two types of disjunctions.

The results for multi-agent disjunctive networks are also highly suggestive. While MaDTNUs provide a high degree of fidelity for modelers, the extreme complexity of deriving a solution makes it an undesirable framework to use in practice. Taking a pragmatic approach, we have two axes against which we can select across multi-agent model. The first axis considers whether we admit disjunctive constraints and the latter considers the fidelity of multi-agent interactions. If we assume a fully observable model of multi-agent uncertainty, our two options are DTNUs and STNUs.

While dynamic controllability can be computed for the former is PSPACE-complete [8], the latter can be determined in $O(n^3)$ time [38]. When we expand our views to include partial observability, we see that while DC-PODTNU has the same computational complexity as dynamic controllability checking for DTNUs, we do not yet know the computational complexity of checking the controllability of POSTNUs or even MaSTNUs. While we do have polynomial time algorithms for checking the dynamic controllability of POSTNUs [10] and MaSTNUs [15], these algorithms are not complete.

It is important to underscore the future importance of investigating the theoretical complexity of dynamic controllability checking for POSTNUs and MaSTNUs. Our work demonstrates that adding partial observability to DTNUs has no impact on the computational complexity of solving the problem, but it is not immediately clear whether the same can be said for STNUs and if they can, whether those benefits continue to hold for full multi-agent networks. While current work has established that certain POSTNUs and MaSTNUs can be checked for controllability in polynomial time, proving a result analogous to the one we present here would significantly expand the set of situations that can be modeled and evaluated efficiently. We believe addressing this question represents an important avenue for future research.

As we look forward, there are still many areas worthy of future research efforts. One of note is the development of novel algorithms for determining the controllability of these networks. Our work establishes bounds on the complexity of computing controllability but does minimal work to provide algorithms for doing so. In practice, our proofs admit the trivial polynomial-space strategy of recursive enumeration of certificates but these algorithms are likely impractical. Our new theoretical bounds open up the challenge of finding novel algorithms that are reasonable for practical use while still respecting polynomial time bounds.

Appendix B

LP Duality and STNs

STNs at their core are systems of linear inequalities, and while it is sufficient to evaluate the system of linear inequalities directly, converting the STN to a distance graph and searching for a negative cycle tends to be quicker in practice. In this appendix, we will attempt to give a stronger intuition for why checking for negative cycles is sufficient. To do so, we consider the linear inequalities more directly and show how the dual LP problem evaluates to exactly the problem of finding a negative cycle in an STN's distance graph.

Because all constraints are in the form of linear inequalities, it is possible to evaluate consistency using a linear program solver (and using an arbitrary objective function). But separately, we can learn more about the underlying structure of STNs and arrive at a more efficient solution method by studying the corresponding LP dual.

We briefly describe the matrix-based representation of the linear inequalities in Figure 2-1a in order to allow us to transform it into the corresponding LP dual. Let x be the vector of events in an STN, and let A be a matrix with each row corresponds to a constraint and each column corresponding to an event.

Each row's values are exactly the linear coefficients of the corresponding constraint; if constraint i is modeled by $u_i \leq x_j - x_k \leq v_i$, then we say that $A_{ij} = 1$, $A_{ik} = -1$, $\forall l \neq j, k : A_{il} = 0$. We let u be the vector of lower-bounds for each constraint i and let v be the vector of upper-bounds.

Given a set of linear inequalities and a desire to find any satisfying solution, it suffices to pass those linear inequalities to an LP solver, setting the objective function to zero:

$$\max 0$$

$$Ax \leq v$$

$$Ax \geq u$$

While it suffices to use an LP solver to evaluate an STN's set of linear equalities and extract a candidate solution, in this instance looking at the LP's dual formulation gives us more information about the nature of the problem and points us towards a more efficient graph-based solution method:

$$\min y_1 \cdot v - y_2 \cdot u$$

$$A^T y_1 - A^T y_2 = 0$$

$$y_1, y_2 \geq 0$$

By strong duality, if we find that the dual is unbounded, we know that the primal is infeasible. If we find a solution whose cost is less than zero, we know that the dual is unbounded (since we can scale any y by a constant, in order to respect the constraint).

In the context of the STN's distance graph, an assignment to y_1 selects a set of edges that have the same direction as the graphical STN constraints, while an assignment of values to y_2 selects from those edges with the opposite direction. The objective function minimizes the total weight of the selected edges, and the constraint enforces that these edges form a cycle. Taken together, the dual is unbounded if and only if there is a negative cycle on the STN's distance graph. In other words, our STN is consistent if and only if there is no negative cycle in its representative distance graph.

Appendix C

Label Reduction Rules

In Chapter 4, we introduced a series of reduction rules, which are reproduced below in Table C.1. In this appendix, we provide a series of lemmas that indicate that these rules are altogether sound.

To simplify our notation, we will use A, B, C, D, and E to represent events. C and E will always refer to contingent events, while A, B, and D may refer to any type of events (either executable or contingent). As such, we provide no guarantee that the edges described in the generation rules come from particular types of constraints. They edges may come from requirement constraints, contingent constraints, or might be byproducts of other edge generation rules.

The first two rules we consider are the *no-case* and *upper-case* rules. These rules

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

Table C.1: Edge generation rules for a labeled distance graph

are identical to edge reduction rules of STNs.

Lemma C.1. ***No-Case Rule***
If we have $A \xrightarrow{u} D$ *and* $D \xrightarrow{v} B$*, then we can add edge* $A \xrightarrow{u+v} B$*.*

Proof. The first two edges correspond to "$D - A \leq u$" and "$B - D \leq v$." Summing the right and left sides of the two constraints, we get that $B - A \leq u + v$. □

Lemma C.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 "$D - A \leq u$". The second provides a conditional constraint "if the contingent constraint ending at C were to take on its maximum possible duration, then $B - D \leq v$".

We proceed with a conditional proof. Assume that the contingent constraint ending at C takes on its maximum possible duration. By modus ponens, we have that $B - D \leq v$. We can sum this result with the constraint $D - A \leq u$ on both sides to get $B - A \leq u + v$. This result holds under the original condition of our proof, implying that the conditional constraint "if the contingent constraint ending at C were to take on its maximum possible duration, then $B - A \leq u + v$" is true. □

Next come the *lower-case* and *cross-case* rules. These are the first rules that strengthen conditional constraints to unconditional ones and the first ones that take into account the effects of delay.

Lemma C.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. Hence, C is not known until after D must be assigned.

For the sake of contradiction, assume that the constraint represented by $A \xrightarrow{x+w} D$ is not always satisfiable, or that it is possible to assign D to occur at some time $t > x + w$ after A occurs. In other words, $D - A > x + w$ We still assume that the original input conditions hold. If we later observe that C happened exactly x units of time after A (or that $C - A = x$), then we have that $D - A - (C - A) > x + w - x$, which simplifies to $D - C > w$ and violates the original constraint, $C \xrightarrow{w} D$. Thus, we must unconditionally enforce the constraint "$D - A \leq x + w$", yielding the edge $A \xrightarrow{x+w} D$ in our labeled distance graph. □

The cross-case rule follows the same logic as the lower-case rule, but in this case, the constraint on $C \rightarrow D$ is conditioned on some other contingent constraint ending at E taking on its maximum possible duration.

In this instance, we are combining two conditional edges, one upper-case and one lower-case, hence, the use of the term cross-case. The original conditional constraint ending at C behaves the same when B takes on its maximum possible value, so the same logic as the lower-case rule applies but with all participating constraints similarly conditioned on B taking on its maximum value. The rule is explained formally in the following lemma:

Lemma C.4. ***Cross-Case Rule***
If we have $A \xrightarrow{c:x} C$, $C \xrightarrow{E:w} D$, $E \neq C$, $C \neq D$, and $w < \gamma(C)$, then we can add edge $A \xrightarrow{E:x+w} D$.

Proof. We can proceed using a conditional proof and assume the antecedent that the contingent constraint ending at E takes on its maximum possible duration. In this case, we can take the upper-case labeled edge $C \xrightarrow{E:w} D$ and rewrite it as $C \xrightarrow{w} D$ since its antecedent condition holds.

We now have two edges $A \xrightarrow{c:x} C$, $C \xrightarrow{w} D$, where $C \neq D$ and $w < \gamma(C)$, so we can immediately apply the lower-case rule and get $A \xrightarrow{x+w} D$.

But because this was a conditional proof, we have that the newly derived consequence only holds when the contingent constraint ending at E takes on its maximum possible duration. To represent this in the edge, we can add the upper-case label E, so we get that our original inputs yield the new rule $A \xrightarrow{E:x+w} D$. □

The final rule is the *label removal* rule. Like the lower-case and cross-case rules, the label removal rule eliminates a label by recognizing that we have to assign values to both events in a constraint before we can determine whether the antecedent represented by the label is true. Whereas in the previous two rules, we eliminated a lower-case label, with the label removal rule, we remove upper-case labels.

Lemma C.5. ***Label Removal Rule***
If we have $B \xrightarrow{C:-u} A$, $A \overset{[x,y]}{\Longrightarrow} C$, $u < x$, *then we can add* $B \xrightarrow{-u} A$.

Proof. In this rule, the first edge tells us that whenever the duration of the contingent link ending at C takes on its maximum value, B happens at most u after A. Since $u < x$, we know that we will not observe 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 the constraint "$B - A \geq -u$" or that $B \xrightarrow{-u} A$ is a valid edge in our distance graph. □

Bibliography

[1] Benjamin Ayton, Nikhil Bhargava, Tiago Vaquero, Eric Timmons, Brian Williams, and Richard Camilli. Risk-bounded goal-directed mission planning for ocean exploration. In *IJCAI'2017 Workshop on Artificial Intelligence in the Oceans and Space*, 2017.

[2] Daniel S Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. The complexity of decentralized control of markov decision processes. *Mathematics of operations research*, 27(4):819–840, 2002.

[3] Nikhil Bhargava, Christian Muise, Tiago Vaquero, and Brian Williams. Delay controllability: Multi-agent coordination under communication delay. In *DSpace@MIT*, 2018.

[4] Nikhil Bhargava, Christian Muise, Tiago Vaquero, and Brian Williams. Managing communication costs under temporal uncertainty. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 84–90. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[5] Nikhil Bhargava, Christian J Muise, and Brian Charles Williams. Variable-delay controllability. In *IJCAI*, pages 4660–4666, 2018.

[6] Nikhil Bhargava, Tiago Vaquero, and Brian Williams. Faster conflict generation for dynamic controllability. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-26, 2017*, pages 4280–4286, 2017.

[7] Nikhil Bhargava and Brian Williams. Multiagent disjunctive temporal networks. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems AAMAS-19*, 2019.

[8] Nikhil Bhargava and Brian C Williams. Complexity bounds for the controllability of temporal networks with conditions, disjunctions, and uncertainty. *Artificial Intelligence*, 271:1–17, 2019.

[9] Lorenz T Biegler and Victor M Zavala. Large-scale nonlinear programming using ipopt: An integrating framework for enterprise-wide dynamic optimization. *Computers & Chemical Engineering*, 33(3):575–582, 2009.

[10] Arthur Bit-Monnot, Malik Ghallab, and Félix Ingrand. Which contingent events to observe for the dynamic controllability of a plan. In *International Joint Conference on Artificial Intelligence (IJCAI-16)*, 2016.

[11] James C Boerkoel and Edmund H Durfee. Distributed reasoning for multiagent simple temporal problems. *Journal of Artificial Intelligence Research*, 47:95–156, 2013.

[12] Jon E Burkhardt and Adam Millard-Ball. Who is attracted to carsharing? *Transportation Research Record*, 1986(1):98–105, 2006.

[13] Massimo Cairo, Carlo Combi, Carlo Comin, Luke Hunsberger, Roberto Posenato, Romeo Rizzi, and Matteo Zavatteri. Incorporating decision nodes into conditional simple temporal networks. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 90. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[14] Massimo Cairo and Romeo Rizzi. Dynamic controllability of conditional simple temporal networks is pspace-complete. In *The 23rd International Symposium on Temporal Representation and Reasoning (TIME)*, pages 90–99. IEEE, 2016.

[15] Guillaume Casanova, Cédric Pralet, Charles Lesire, and Thierry Vidal. Solving dynamic controllability problem of multi-agent plans with uncertainty using mixed integer linear programming. In *ECAI*, pages 930–938, 2016.

[16] Alessandro Cimatti, Luke Hunsberger, Andrea Micheli, Roberto Posenato, and Marco Roveri. Sound and complete algorithms for checking the dynamic controllability of temporal networks with uncertainty, disjunction and observation. In *The 21st International Symposium onTemporal Representation and Reasoning (TIME)*, pages 27–36. IEEE, 2014.

[17] Alessandro Cimatti, Andrea Micheli, and Marco Roveri. Dynamic controllability of disjunctive temporal networks: Validation and synthesis of executable strategies. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 3116–3122, 2016.

[18] Carlo Combi, Luke Hunsberger, and Roberto Posenato. An algorithm for checking the dynamic controllability of a conditional simple temporal network with uncertainty. *Evaluation*, 1:1, 2013.

[19] Carlo Comin and Romeo Rizzi. 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)*, pages 19–28. IEEE, 2015.

[20] Patrick R Conrad and Brian Charles Williams. Drake: An efficient executive for temporal plans with choice. *Journal of Artificial Intelligence Research*, 42:607–659, 2011.

[21] Johan De Kleer and Brian C Williams. Diagnosing multiple faults. *Artificial intelligence*, 32(1):97–130, 1987.

[22] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial intelligence*, 49(1-3):61–95, 1991.

[23] Robert Effinger, Brian Williams, Gerard Kelly, and Michael Sheehy. Dynamic controllability of temporally-flexible reactive programs. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling*, Thessaloniki, Greece, September 2009.

[24] Pavlos Eirinakis, Salvatore Ruggieri, K Subramani, and Piotr Wojciechowski. On quantified linear implications. *Annals of Mathematics and Artificial Intelligence*, 71(4):301–325, 2014.

[25] Cheng Fang, Peng Yu, and Brian C Williams. Chance-constrained probabilistic simple temporal problems. 2014.

[26] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[27] Philip E Gill, Walter Murray, and Michael A Saunders. Snopt: An sqp algorithm for large-scale constrained optimization. *SIAM review*, 47(1):99–131, 2005.

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

[29] Luke Hunsberger and Roberto Posenato. 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)*, pages 175–183, 2016.

[30] Luke Hunsberger, Roberto Posenato, and Carlo Combi. The dynamic controllability of conditional stns with uncertainty. *arXiv preprint arXiv:1212.2005*, 2012.

[31] Michel Donald Ingham. *Timed model-based programming: Executable specifications for robust mission-critical sequences.* PhD thesis, Massachusetts Institute of Technology, 2003.

[32] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 302–311. ACM, 1984.

[33] Erez Karpas, Steven James Levine, Peng Yu, and Brian C Williams. Robust execution of plans for human-robot teams. In *International Conference on Automated Planning and Scheduling*, pages 342–346, 2015.

[34] Phil Kim, Brian C Williams, and Mark Abramson. Executing reactive, model-based programs through graph-based temporal planning. In *International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 487–493, 2001.

[35] Steven Levine and Brian Williams. Concurrent plan recognition and execution for human-robot teams. In *ICAPS-14*, 2014.

[36] Michael D Moffitt. On the partial observability of temporal uncertainty. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 1031. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

[37] Paul Morris. A structural characterization of temporal dynamic controllability. In *International Conference on Principles and Practice of Constraint Programming*, pages 375–389. Springer, 2006.

[38] Paul Morris. Dynamic controllability and dispatchability relationships. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 464–479. Springer, 2014.

[39] Paul Morris and Nicola Muscettola. Managing temporal uncertainty through waypoint controllability. In *International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 1253–1258, 1999.

[40] Paul Morris and Nicola Muscettola. Temporal dynamic controllability revisited. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 1193–1198, 2005.

[41] Mikael Nilsson, Jonas Kvarnström, and Patrick Doherty. Incremental dynamic controllability revisited. In *Proceedings of the International Conference on International Conference on Automated Planning and Scheduling (ICAPS-2013)*, pages 337–341, 2013.

[42] Mikael Nilsson, Jonas Kvarnström, and Patrick Doherty. Incremental dynamic controllability in cubic worst-case time. In *21st International Symposium on Temporal Representation and Reasoning (TIME-2014)*, pages 17–26. IEEE, 2014.

[43] C.H. Papadimitriou. Computational complexity. *Addison-Wesley Reading*, 1994.

[44] Bart Peintner, Kristen Brent Venable, and Neil Yorke-Smith. Strong controllability of disjunctive temporal problems with uncertainty. In *International Conference on Principles and Practice of Constraint Programming*, pages 856–863. Springer, 2007.

[45] Léon Planken. Temporal reasoning problems and algorithms for solving them. 2007.

[46] Pedro Santana, Tiago Vaquero, Claudio Fabiano Motta Toledo, Andrew J. Wang, Cheng Fang, and Brian C. Williams. Paris: a polynomial-time, risk-sensitive scheduling algorithm for probabilistic simple temporal networks with uncertainty. In *Proceedings of the 26th International Conference on Automated Planning and Scheduling*, 2016.

[47] Julie A Shah, John Stedl, Brian C Williams, and Paul Robertson. 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)*, pages 296–303, 2007.

[48] John Stedl and Brian C Williams. A fast incremental dynamic controllability algorithm. In *Proceedings of the ICAPS Workshop on Plan Execution: A Reality Check*, pages 69–75, 2005.

[49] Kostas Stergiou and Manolis Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence*, 120(1):81–117, 2000.

[50] Larry J Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976.

[51] K Subramani. On a decision procedure for quantified linear programs. *Annals of Mathematics and Artificial Intelligence*, 51(1):55–77, 2007.

[52] Teruo Sunaga. Theory of interval algebra and its application to numerical analysis. *RAAG memoirs*, 2(29-46):209, 1958.

[53] Eric Timmons, Tiago Vaquero, Brian Williams, and Richard Camilli. Preliminary deployment of a risk-aware goal-directed executive on autonomous underwater glider. In *PlanRob Workshop, ICAPS 2016*, 2016.

[54] Ioannis Tsamardinos, Thierry Vidal, and Martha E Pollack. Ctp: A new constraint-based formalism for conditional, temporal planning. *Constraints*, 8(4):365–388, 2003.

[55] Kristen Brent Venable, Michele Volpato, Bart Peintner, and Neil Yorke-Smith. Weak and dynamic controllability of temporal problems with disjunctions and uncertainty. In *Workshop on constraint satisfaction techniques for planning & scheduling*, pages 50–59, 2010.

[56] Kristen Brent Venable, Michele Volpato, Bart Peintner, and Neil Yorke-Smith. Weak and dynamic controllability of temporal problems with disjunctions and uncertainty. In *Workshop on Constraint Satisfaction Techniques for Planning & Scheduling*, pages 50–59, 2010.

[57] Kristen Brent Venable and Neil Yorke-Smith. Disjunctive temporal planning with uncertainty. In *International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 1721–1722, 2005.

[58] Thierry Vidal and Helene Fargier. Handling contingency in temporal constraint networks: from consistency to controllabilities. *Journal of Experimental & Theoretical Artificial Intelligence*, 11(1):23–45, 1999.

[59] Andrew J. Wang and Brian C. Williams. Chance-constrained scheduling via conflict-directed risk allocation. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, January 2015.

[60] Brian C Williams and Robert J Ragno. Conflict-directed a* and its role in model-based embedded systems. *Discrete Applied Mathematics*, 155(12):1562–1595, 2007.

[61] Peng Yu. *Collaborative Diagnosis of Over-Subscribed Temporal Plans*. PhD thesis, Massachusetts Institute of Technology, October 2016.

[62] Peng Yu, Cheng Fang, and Brian C Williams. Resolving uncontrollable conditional temporal problems using continuous relaxations. In *ICAPS*, 2014.

[63] Peng Yu and Brian C Williams. Continuously relaxing over-constrained conditional temporal problems through generalized conflict learning and resolution. In *International Joint Conference on Artificial Intelligence (IJCAI-13)*, pages 2429–2436, 2013.

[64] Matteo Zavatteri. Conditional simple temporal networks with uncertainty and decisions. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 90. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.