Distributed Multi-Agent Decision Making Under Uncertain Communication

by

Cameron W. Pittman

M.A., Belmont University (2011)

B.A., Vanderbilt University (2009)

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Master of Science in Aeronautics and Astronautics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2023

Author .....

Department of Aeronautics and Astronautics

August 15, 2023

Certified by .....

Brian C. Williams

Professor of Aeronautics and Astronautics, MIT

Thesis Supervisor

Accepted by .....

Jonathan P. How

R.C Maclaurin Professor of Aeronautics and Astronautics, MIT

Chair, Graduate Program Committee

Distributed Multi-Agent Decision Making Under Uncertain Communication

by

Cameron W. Pittman

Submitted to the Department of Aeronautics and Astronautics
on August 15, 2023, in partial fulfillment of the
requirements for the degree of
Master of Science in Aeronautics and Astronautics

Abstract

As space exploration accelerates and the number of robots and humans working in extreme environments grows with it, we must enact autonomous multi-agent coordination in order to safely operate in environments that are inherently hostile to communication. To the best of our knowledge, there are no autonomous executives capable of coordinating with other agents while independently reasoning over communication delay to decide when to act. A key gap that must be addressed is a single-agent executive capable of deciding when to act given communication delay, which can the form the basis for a multi-agent execution context. Existing research has provided insights into temporal reasoning, namely modeling observation delay and scheduling actions with temporal constraints, but there is both a need for deciding when to schedule events when there is uncertain observation delay, and a need to robustly coordinate between agents. Scheduling actions in the face of uncertainty is a challenge due to the compounding uncertainties of uncontrollable exogenous events, unknown observation delay, and uncertain communication between agents. This thesis puts forth a series of contributions that culminates in a robust single-agent executive demonstrated to run on real hardware with the ability to coordinate in a multi-agent context despite observation delay. Doing so required insights in checking variable-delay controllability, defining an execution strategy against temporal constraints with variable observation delay, and online coordination for multiple agents. We show that single agent online scheduler exhibits the expected performance characteristics, and demonstrate multi-agent execution with uncertain communication in the context of a simulated astronaut and robotic asset collaborating with communication delay.

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

Acknowledgements

My journey to this SM thesis began eight years ago when I was a software engineer living in San Francisco. Two years prior, I had just wrapped four years of teaching high school science and didn't know how to code. I taught myself Python using MIT OpenCourseware on a whim because it was something I had always wanted to learn. Coding took over my life the first time I wrote a function. My first projects were physics demos and websites, but soon I learned I loved helping systems make decisions. And there is absolutely no system cooler than human spaceflight. So I sat there at my desk in SF and wondered, "I want to be a part of the next space race! What if I studied autonomy for human spaceflight? Where would I even go to learn autonomy?" (I didn't even know the field was called "autonomy" at the time.) Of course, MIT came up when I started searching. It took five more years of studying computer science, finding people to learn from, and integrating in the human spaceflight community, but eventually I found my way into AeroAstro and the MERS lab. It's been the journey of a lifetime and there's no way I would be here without the love and support from so many people.

First, I have to thank my amazing wife, Mo, who is the hardest working, most driven person I've ever met. She's been my biggest cheerleader and sounding board for my ideas (even when she has no idea what I'm talking about). Our newborn daughter, Catalina, is my inspiration. I think about her living in a world where there are thousands of people living and working in space. At the time of writing this, she's a month old and already making me work harder than I ever have. I never would have made it this far without a supportive family. My parents, Mark and Suzanne, and my brother, Max, have always been there for me.

My advisor, Brian Williams, has consistently surprised me with how much support he's freely given. Brian has been one of the most helpful people I've ever met, starting with the first time I introduced myself and blurted out, "hi, I'm taking your class and I think it's amazing and I want to use everything at work and can I please join your lab?" From academic mentoring, to brainstorming executive architectures, to

explaining basic concepts in temporal reasoning, to envisioning the next generation of online education, he's been open, honest, and eager to share ideas. Brian, you have had a profound impact on my life and career, and for that I'm forever grateful.

Next, I have to thank all my labmates. I can't really quantify the level of impact any single person has had, so I'll be doing these chronological order of when we met.

The first people I met were my 16.413 TAs, Simon Fang and Sungkweon Hong. I would go to office hours even when I didn't have questions just because I wanted to hang out and talk autonomy with people. Simon, thank you for being a supportive TA (and introducing me to *Aunty Donna*). Sungkweon, ...

Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of source codes

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

Chapter 1

Introduction

The United States, Europe, private companies, and new space agencies around the world are collectively urging humanity further into the cosmos with a sense of urgency that has not been seen since Apollo. We are witness to the great forces of political will, economics, and technological prowess launching our foothold beyond low Earth orbit and into planetary colonization. From every perspective, the scale of space exploration is staggering. It will take relentless effort from untold numbers of people, and our robotic partners, to usher us into the era where interplanetary travel and living off-world are commonplace. In every envisioned scenario for deep space exploration, there is a need to coordinate between agents, whether human or robotic, who must find a way to safely work together in the face of the communication challenges inherent to extreme environments.

There is good reason to design a system for coordination around the notion of uncertain communication. We take for granted that communication is easy in our civilized corners of the Earth's surface. Cellular signals and WiFi are security blankets, tricking us into thinking it must be easy for *everyone* to communicate *everywhere*. The fact of the matter is that the communication is far from a given when you leave civilization. Consider low Earth orbit. The largest artificial satellite, the International Space Station (ISS), has been in orbit since 1998. Despite that, it still loses communication with the ground regularly. Satellites with much less robust infrastructure lose contact with the ground even more often. When astronauts set foot

on the South Pole of the Moon soon as part of the Artemis program,[1] they will find uncertain satellite coverage and the need to contend against local topology that is hostile to radio signals [1]. Any robots working near a habitat, or any astronauts collecting samples nearby, may rely on complicated systems of relays that can provide, at best, unpredictable communications with nearby *in situ* agents, let alone Mission Control on Earth. Communications may be delayed due to passing through multiple relays and the speed of light. Or communications may drop out altogether when agents accidentally walk behind a big boulder between them and the closest relay.

The Artemis astronauts will continue a longstanding tradition in the U.S. space program of performing Extravehicular Activities (EVAs) [2]. Like the Apollo astronauts of the 1960s and 1970s, Artemis astronauts will embark on EVAs wherein crews don spacesuits, egress landers, and conduct scientific expeditions on the lunar surface. There, they will survey surface features, collect samples, and in general perform field geology [3][5]. A small number of astronauts are Ph.D. geologists by trade, and NASA is training others in the principles of field geology up to a notional masters level of understanding [6]. NASA will support the lunar activities of these astronauts through a vast infrastructure of personnel on the ground, including teams of domain-relevant scientists. Together with flight controllers and engineers from various disciplines, a science team on the ground will provide real-time feedback to ensure that Artemis astronauts maximize the scientific return of their EVAs.

Throughout an EVA, the crew (astronauts with spacesuits on), need to know what to do. Currently, Mission Control Center (MCC or simply "ground"), a vast infrastructure with hundreds of personnel, monitors all aspects of EVAs and provides timely guidance to the crew. A key role MCC plays is keeping the crew on schedule by informing them as to what actions need to be taken and when to take them [7]. Current operations use a mix of manual processes and automated tools[2] for tracking a crew's progress on an EVA timeline. Crews then rely on verbal communications with MCC, as well as written notes kept on their suits (e.g. Figures 1-1 and 1-2), to

[1] `https://www.nasa.gov/specials/artemis/`

[2] E.g. `https://github.com/nasa/maestro`

perform the right actions at the right times. It is not a stretch to imagine that future spacewalkers will don new spacesuits that feature digital assistants designed to walk them through procedures. Astronauts may interact with digital assistants through heads up displays, over voice, or on wrist mounted screens. If communications with MCC are challenged, then behind a digital assistant will need to be a system capable of providing the same timely advice that MCC provides today. To do so, it will need to reason over all the actions in an EVA timeline, while taking into account the current state of what has been done and what needs to be done, in order to help the crew decide when actions should be performed. We call such a system a *scheduler*.

Figure 1-1: Astronaut Alan Bean on a lunar EVA during Apollo 12. The photographer, astronaut Pete Conrad, can be seen in Bean's helmet. Note Bean's left wrist, which sports a printed EVA timeline with notes about the tasks the crew needed to perform. *Credit: Charles "Pete" Conrad, Apollo 12, NASA.*

Of course, astronauts never work alone. In addition to communicating with MCC,

Figure 1-2: Astronaut Peggy Whitson on an ISS EVA in 2017. Note her left wrist, which sports a printed EVA timeline with notes about the tasks the crew needed to perform. *Credit: NASA (source).*

crews perform EVAs on a buddy system, meaning two crew members egress and ingress at the same time. At any given moment, they may be collocated, distant, working independently, or in tight cooperation. Robotic assets also play a role. On ISS, crews work with manipulators like the Canadarm [8]. Rovers and robotic systems will almost certainly be present on the Moon. They may *collaborate* in many ways. Fundamentally, collaboration is a process of deciding when to perform actions based on when other agents acted. For instance, taking a picture is a form of collaboration. In Figure 1-1, Pete Conrad clearly decided when to snap a photo by observing when Alan Bean held up the sample container. Maybe Conrad saw Bean getting ready and knew when to act. Or, more likely, Bean told Conrad over the radio that he was ready for a picture, so Conrad should get the camera ready. We envision that future digital assistants must facilitate similar collaboration. To do so, each agent will have their own assistant, each running a scheduler that can reason over timing constraints between their actions and the actions of their peers.

The gap in the current state of the art is that, to the best of our knowledge, there is no such scheduler that is capable of reasoning over the sources of uncer-

US EVA 22 SUMMARY TIMELINE

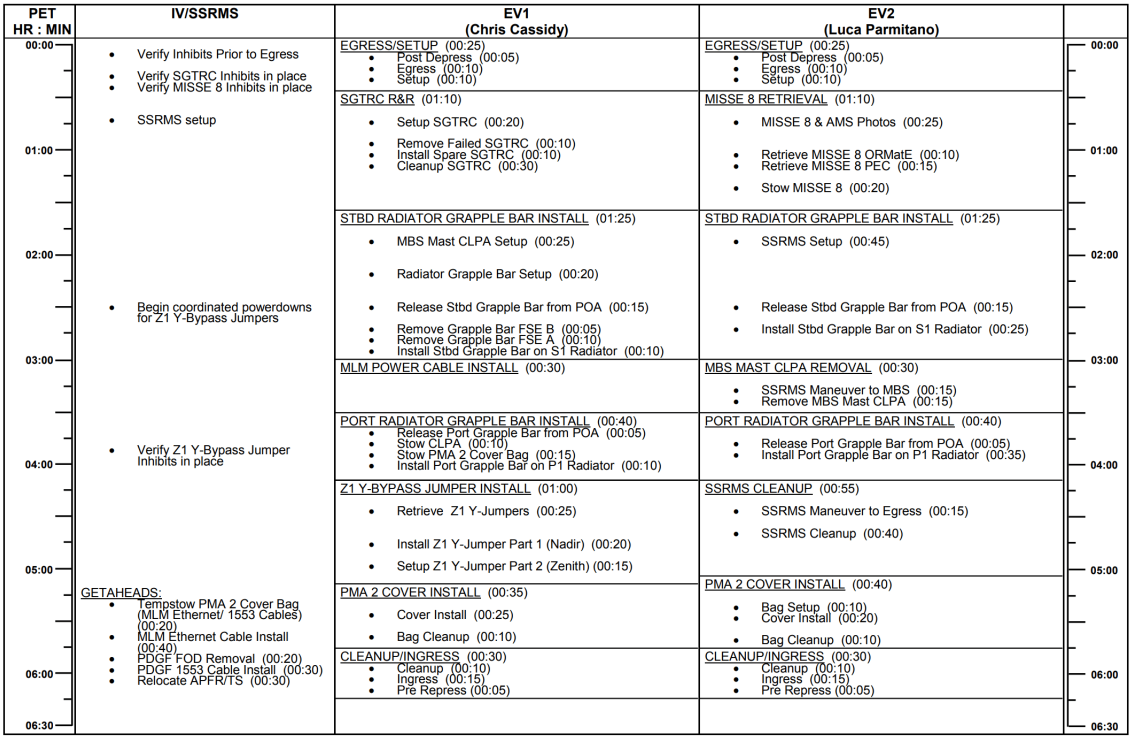


Figure 1-3: The summary page from US EVA 22 on ISS. Each column represents a different agent with time increasing from top to bottom. "PET" is the Phased Elapsed Time, or time since the EVA began. Each activity has a time in HH:MM seconds associated with it. SSRMS is the Canadarm. EV1 and EV2 are the two spacewalkers. *Credit: NASA (source)*

tain communication and observation delay. Existing schedulers assume instantaneous communications, which is unreasonable on the lunar surface, or indeed, any extreme environment. Furthermore, state of the art schedulers do not provide collaborative capabilities that allow multiple agents to work together, especially in the presence of uncertain communication. This thesis leverages and contributes to temporal reasoning research to implement such a scheduler for multi-agent (MA) collaboration with uncertain communication, which we refer to as a *delay scheduler*.

For the remainder of the introduction, we present a short summary of each chapter.

1.1 Problem Statement

A delay scheduler can be used in the case of both one agent (e.g. a single astronaut or a robot) working individually, as well as when multiple agents are collaborating. We start by defining the problem statement for the single-agent case, before identifying the features that are necessary for the MA case.

We use tools from temporal reasoning, namely temporal networks [9], to model EVA timelines as constraints (relationships) between a finite set of events. Some events are under an agent's control, while others are not. Some events may have associated uncertain observation delay.

At some time t during an EVA, we have a set of events that were *observed* before t. When an event has been recorded at t, we say that it has been *assigned*. If there is no associated observation delay with an event, then the time of an observation is the same as assignment. If there is associated observation delay, then it is possible that assignment times are earlier than their respective observations.

We want a *Real-Time Execution Decision* (RTED), which consists of unexecuted events and when they should be performed. Each RTED consists of a set of unexecuted events to be scheduled at a future time.

Our problem statement for the delay scheduler is as follows. For some time t during scheduling, the delay scheduler should take a temporal network, observation delay, observations thus far, and assignments so far as input. It should output an

RTED.

We expand the previous problem statement to the multi-agent case by adding the notion of agents. Each agent has their own delay scheduler. As such, each delay scheduler receives a temporal network. A subset of all events in their temporal network will be received from their peers in the form of communications. From the perspective of an agent, their peers simply need to be aware of what events have been assigned. Events that an agent receives from peers are no different than observations.

To be clear, each agent needs their own temporal network. In the EVA domain, we see the same separation in Figure 1-3 where each agent in the EVA has its own set of actions to perform. While some actions are aligned between agents, there is no assumption that all agents are working against the same events with the same constraints.

Thus, the multi-agent addition to the problem statement follows. Given event assignments and a set of peers, all assignments made at t should be immediately communicated to all peers.

1.2 Approach

The architecture of the delay scheduler is designed around the notion of taking everything we know about set of temporal constraints and when events have been assigned and distilling it down to a single RTED. There are three key processes in the delay scheduler.

1. an offline process that initializes the scheduler with a given model, including the temporal network and observation uncertainty,
2. an online process that updates the scheduler with event assignments,
3. an online process that broadcasts event assignments with peers, and
4. an online process that queries for RTEDs.

Much like how a flight controller cannot provide guidance on an EVA timeline without an accurate copy of the EVA timeline, before scheduling begins, the scheduler

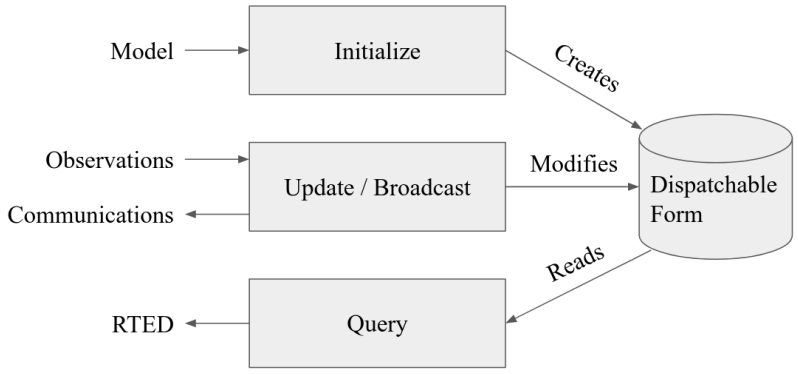


Figure 1-4: A sample architecture with two delay schedulers collaborating. Each agent receives a single temporal network as input. Observations of the outside world are recorded. Communications relay event assignments to peers. Each agent outputs its own RTED.

must be given a model of the schedule. Such a model will be unique to a given agent and must include all events, the constraints between events, and the observation delay associated with events. Figure 1-4 represents this input as a separate model given to each scheduler offline.

During scheduling, schedulers receive observations of events. For a given agent, a, observations come from three sources: actions a has sensed but not controlled ("I traversed difficult terrain and reached the installation location at $t = 5$"), actions a has performed ("I put the tripod down at $t = 6$"), and actions that have been communicated to a ("It is $t = 15$ and my peer told me their confirmation arrived"). Figure 1-4 shows observations coming from outside the two schedulers while communications are passed between them.

As scheduling progress, a digital assistant will want to ask the scheduler for guidance as to when to act. We represent the delay scheduler's output as an RTED, which can also be seen in Figure 1-4.

With three distinct processes involved in single-agent scheduling (subprocesses 1, 2, and 4), we naturally define three explicit interfaces on the delay scheduler. Figure 1-5 shows the flow of information between the interfaces and introduces a new data structure called the *dispatchable form*. In the context of scheduling, the dispatchable form is a graph structure that acts like a database. Event assignments are recorded

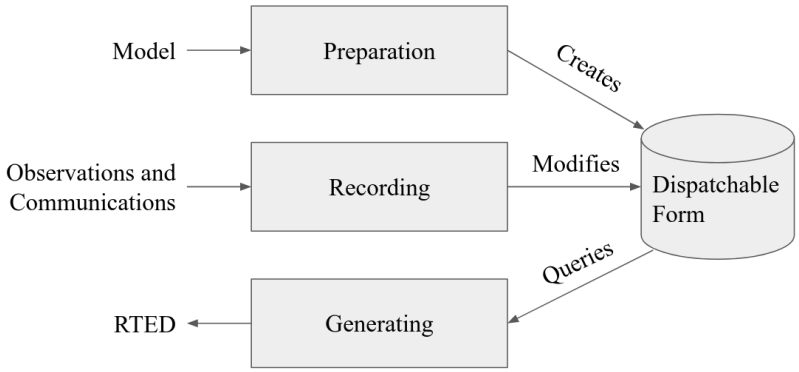


Figure 1-5: The four interfaces of a delay scheduler. The first is for initialization, the second is for schedule updates, the third is for broadcasting, and the fourth is for generating RTEDs. The dispatchable form is ultimately the source of truth for when events should be scheduled.

to the dispatchable form, and the dispatchable form can be queried to find the next RTED. Note that Figure 1-5 is a simplification. The dispatchable form is the key data structure that make scheduling possible, but an implementation of a scheduler will store forms of data other than the dispatchable form when events are recorded.

Importantly, the second interface in Figure 1-5, recording, takes both observations and communications as input. A key idea for the delay dispatcher is that communications from agents are no different than event observations. Peer schedulers communicate when they have assigned events, which then received as observations. We shall see that equating communications and observations allows us to define a single-agent delay scheduler that can be seamlessly integrated in a multi-agent context.

To enable communications between agents, delay schedulers are networked such that there is a communication pathway between all agents. We do not assume every agent can communicate with ever peer, rather when communications are received, they are relayed to all known peers.

1.3 Modeling Temporal Constraints with Uncertain Observations

As stated before, our choice for modeling the "what" and "when" of scheduling is a temporal network, also called a temporal constraint network [9].

Temporal networks consist of events and constraints. Written in English, events and constraints might be stated as "samples must be stowed no more than five minutes after being collected." In this case, `sample-collecting` and `sample-stowing` would be two events. It is the case that mission planners have a robust set of modeling tools for creating schedules. In the literature, there are constraints between events we can control [9], events we cannot control [10], constraints between multiple agents [11], events that may not be observed [12], and events with variable observation delay [13]. We highlight key components of our chosen modeling framework below.

Our choice of modeling constraints is set-bounded ranges. That is, a constraint between two events, "sample collecting" and "sample stowing" is represented as `sample-collecting` $\xrightarrow{[0,5]}$ `sample-stowing`. `sample-stowing` must be scheduled no earlier than 0 time units beforee and no later than 5 time units after `sample-collecting`. This constraint assumes both `sample-collecting` and `sample-stowing` are under the astronaut's full control. Perhaps the astronauts are working separately with one sample collection bag shared between them. In that case, an astronaut might need to wait for their buddy to finish using the bag before stowing samples. If so, then `sample-stowing` is outside their control. We would then model the constraint as, say, `sample-collecting` $\xRightarrow{[0,5]}$ `sample-stowing`. Now, the constraint dictates that `sample-stowing` will happen no later than five minutes after sample collection, but the astronaut cannot choose (control) when in the next five minutes `sample-stowing` is scheduled.

Our choice of model for uncertain communication is variable observation delay [13]. Say there is uncertain communication between the astronauts. Now the communication indicating that `sample-stowing` can be begin may arrive either immediately, or a minute after it was sent. We model the delay using a *variable-delay function*,

$\bar{\gamma}(\texttt{sample-stowing}) = [0, 1]$. Altogether, the astronaut may receive the communication indicating that `sample-stowing` may begin instantaneously, or with up to a minute of delay. Key to our model is that the receiver *does not know how much a message was delayed*. If the message is received at $t = 4$, then the communication may have been sent at $t = 4$ and received immediately. Or it is possible that it was sent as early as $t = 3$ and received after a minute delay.

Temporal networks play two key roles in scheduling. First, they allow a modeler to represent the events and constraints between events of a schedule in a form that a scheduler can ingest. Second, they can be checked for *controllability* (also called *consistency*). In order for it to be possible for a temporal network to be scheduled, there must be a set of assignments for all events under the agents control that satisfies all constraints in spite of the fact that some events may arrive late or never at all.

With our choice of modeling constraints with variable observation delay, we perform a *variable-delay controllablity* check on temporal networks passed to the delay scheduler. A key aspect of checking controllability is that we must the temporal network to one with less uncertainty that is equivalent with respect to controllability. It is this less uncertain form of the temporal network that we then schedule.

1.4 Scheduling Events Despite Uncertain Observations

From this point forward, we assume that the scheduler has been given a controllable temporal network that accurately models the world.

Other researchers have presented single-agent scheduling algorithms for temporal networks with uncontrollable events [14], [15], the fastest being FAST-EX [15]. An underlying assumption of existing schedulers is that events are observed instantaneously, that is $\texttt{obs}(x_c) = \xi(x_c)$ for some uncontrollable event x_c. Events with uncertain observations are incompatible with this assumption, necessitating a change to the way observations are recorded. The delay scheduler is a modified version of

FAST-EX.

In fact, there are broadly two key differences between a delay scheduler and a scheduler that implements FAST-EX. First, we must account for observation delay when events are observed. For instance, if we know an observation at time t was delayed by γ time units, the assigned time is then $t - \gamma$. Second, we introduce a new variable to RTEDs, a *no-operation*, or `no-op`, boolean. Some events in an RTED may be `no-op` for reasons explained below.

When we transform the original temporal network with uncertain observations to one with less uncertainty, we artificially shrink some of the constraints in the original temporal network. Some uncontrollable events may arrive earlier or later than expected. We address these situations with *buffering* and *imagining* uncontrollable events. We use the `no-op` addition to RTEDs to simplify the handling of events that must be buffered or imagined.

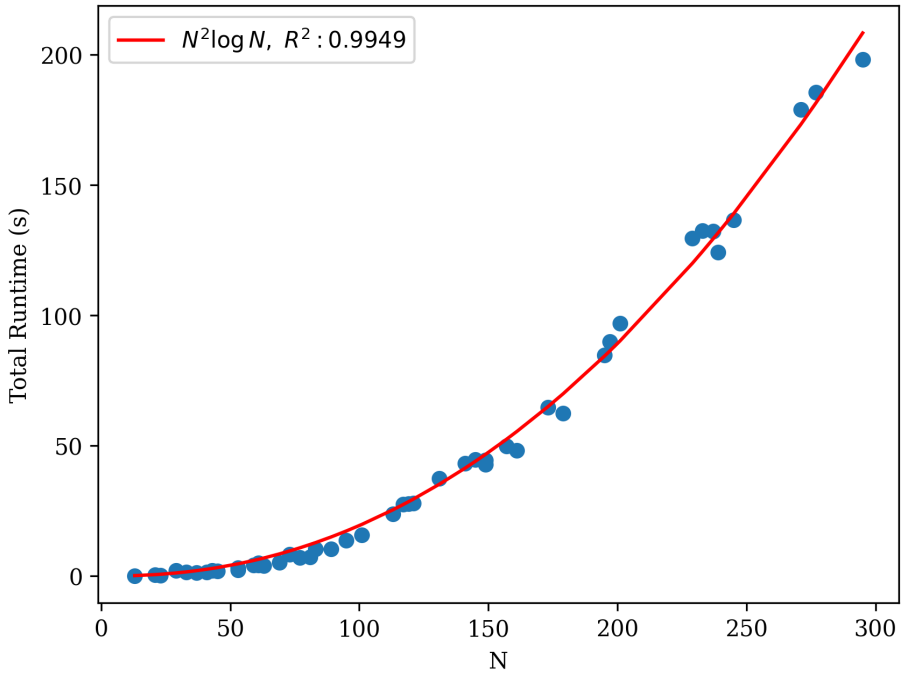


Figure 1-6: Total runtime data for scheduling all events in temporal networks with uncertain observations with less than 300 events.

We demonstrate that the delay scheduler exhibits the performance characteristics of FAST-EX. At the core of FAST-EX is a Dijkstra Single Sink/Source Shortest

Paths subroutine, which limits the runtime performance. Each call to the subroutine should have a runtime performance of $O(N \log N)$, where N is the number of events in the temporal network. Thus, we expect the total runtime to schedule all events in a temporal network to be $O(N^2 \log N)$. To evaluate the performance of the delay scheduler, we scheduled randomly generated temporal networks with a structure inspired by a satellite dish installation procedure. In the experiments, we model multiple astronauts (up to eight) working in parallel with inter-agent temporal constraints. Figure 1-6 shows that the delay scheduler demonstrates the expected performance characteristics against said temporal networks.

1.5 An Envisioned Executive for Dispatching Actions with Uncertain Observations

We need a means to connect the RTEDs of a delay scheduler with the actions an agent performs. We envision that the delay scheduler can serve as the scheduling logic behind an astronaut's digital assistant, or in the case of a robot, a *task executive*. A task executive should allow a human modeler to provide constraints as input. The task executive is then charged with generating a plan and dispatching actions as output.

We integrate the delay scheduler into a high-level task planner known as *Kirk*. We call our variant of Kirk, *Delay Kirk*. A simplified overview of Delay Kirk's architecture can be found in Figure 1-7. Delay Kirk takes the Reactive Model-Based Programming Language (RMPL) [16], a high-level language for modeling hybrid automata and constraints, as input. It then creates a temporal plan network and chooses timed actions to execute to satisfy all the goals as specified in RMPL. It is at this point that the delay scheduler can be integrated into delay Kirk. With events and temporal constraints between them, the delay scheduler can produce RTEDs and tell Delay Kirk when to act.

For the purpose of this thesis, planning is out of scope. Instead, we focus on the

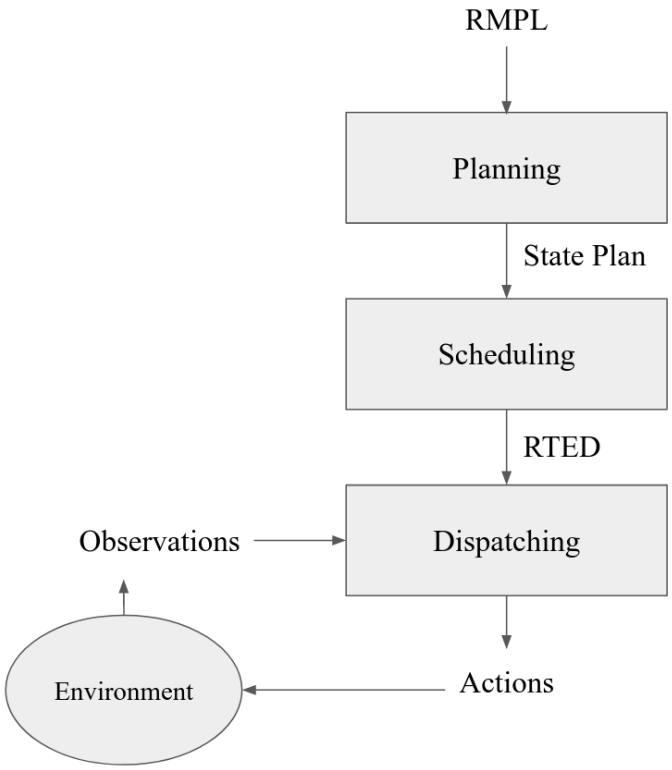


Figure 1-7: A high-level overview of the Delay Kirk task executive architecture with respect to dispatching actions.

delay dispatcher a component that enables an executive to impact an environment by taking actions based on the RTEDs the delay scheduler produces.

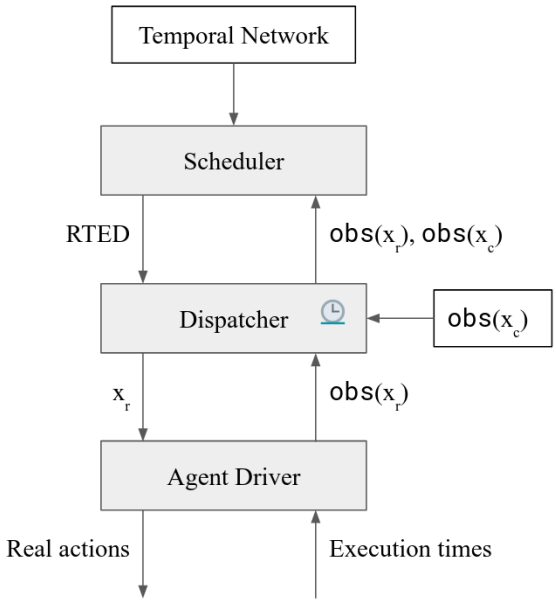


Figure 1-8: A more detailed view of the delay dispatcher architecture.

In Figure 1-8, we introduce a new component, the *driver*. We also define new variables in order to paint a complete picture of the role the delay dispatcher plays. x_r represents a controllable event. $\texttt{obs}(x_r)$ and $\texttt{obs}(x_c)$ represent the times that controllable and uncontrollable events are observed respectively. The actions that the dispatcher dispatches are mediated through the driver. Essentially, it translates events to commands that cause actions to happen in the real-world. For a digital assistant, a driver might send a command to update the heads-up-display in the crew's helmet. For a robot, the driver might publish a ROS message [17].

Observations are passed to the scheduler through the dispatcher. We do so because in our architecture the dispatcher, not the scheduler, has access to a clock. The dispatcher takes the responsibility of comparing RTEDs to the clock time and deciding when to act. Likewise, when events are observed, the dispatcher tells the scheduler when they were observed. This change has the cumulative effect of giving the dispatcher responsibility for interacting with the environment.

A key distinction between a dispatcher for instantaneous observations and the

delay dispatcher is that not all observed events are scheduled immediately. It is the case that some observations must be buffered to a later time to be scheduled. If so, the dispatcher has the responsibility of actually waiting until the correct clock time to record the time in the scheduler.

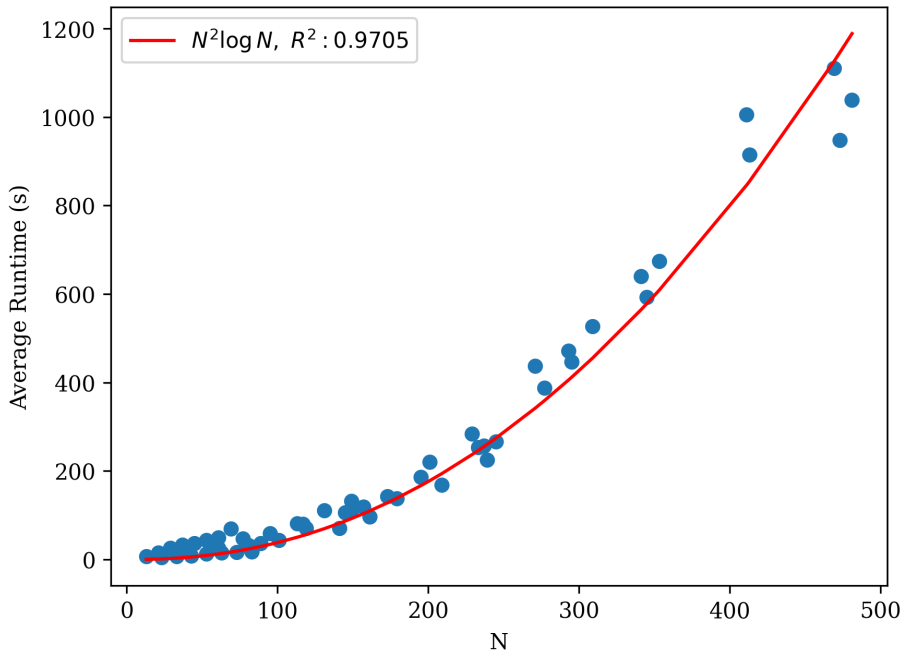


Figure 1-9: A comparison of the total runtime to run the dispatcher against the number of events in a temporal network.

We evaluate the dispatcher's interface that loops and compares the clock time to an RTED to decide when to act. For this tests, we use the same randomly generated temporal networks as were used when evaluating the scheduler. Figure 1-9 shows the total runtime for all calls to the dispatcher while scheduling all events in a temporal network. The Dijkstra updates that are performed when recording events dominates the runtime performance. Given every event is recorded inside this loop, we see the same $O(N^2 \log N)$ performance we saw when looking at the total time to run all schedule updates.

1.6 Multi-Agent Scheduling with Uncertain Observations

Collaboration between agents is enabled by enforcing that communications between agents are treated the same as uncontrollable event observations. Thus, we are only challenged to define communication pathways between agents that guarantee agents receive relevant observations. We do so by networking delay schedulers in a *communication graph*. A communication graph is a simple directed graph that is used to broadcast event propagation messages between peers.

We evaluate the multi-agent delay scheduler in two simulations. In the first simulation, we run three instances of Delay Kirk with inter-agent constraints between them. We compare their schedules to the same schedule that would be produced if one delay scheduler tried to schedule all events for all three agents. We found that the multi-agent delay dispatchers were able to schedule events while respecting all inter-agent constraints.

We finally present a hardware demonstration with a Barrett WAM manipulator being controlled by one Delay Kirk, with another Delay Kirk representing an astronaut's digital assistant. We demonstrate that Delay Kirk is able to dispatch actions to the Barrett WAM while receiving communications representing inter-agent constraints over HTTP.

1.7 Thesis Structure

The structure of this thesis is as follows. A more detailed problem statement, including descriptions of the scenarios used for testing distributed collaboration and coordination with uncertain communication, will be provided in Chapter 2. Our approach to addressing the problem statement will be outlined in Chapter 3. Chapter 4 will provide the first technical contributions of this thesis, first by addressing the issue of modeling observation delay, then by providing a procedure that can be used to guarantee that temporal constraints with observation delay are satisfiable.

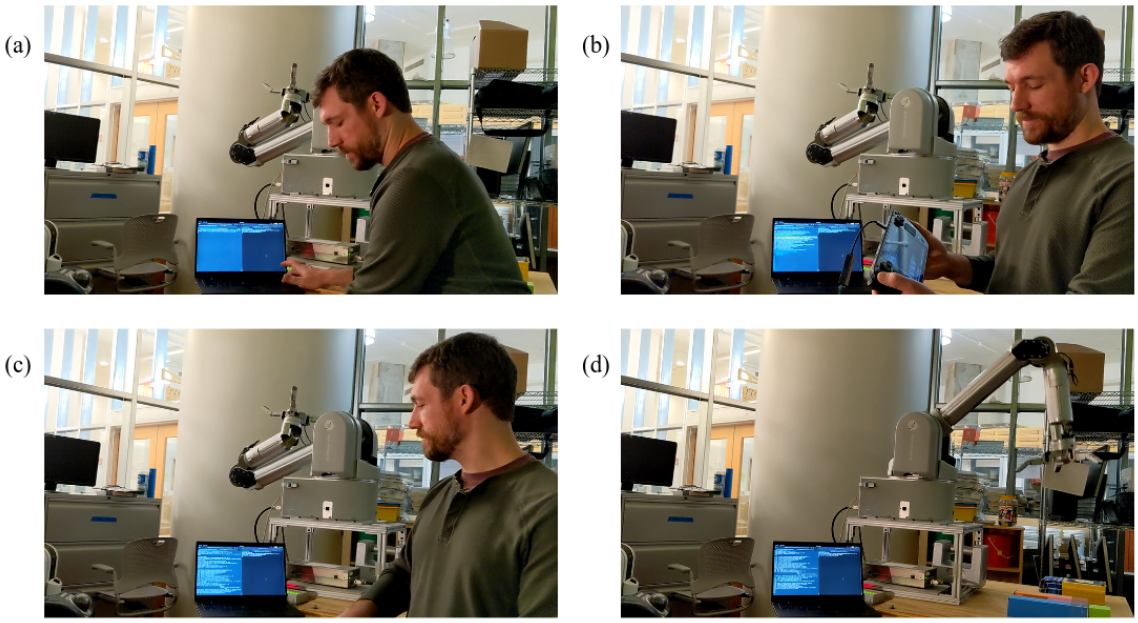


Figure 1-10: A hardware demonstration in four parts. (a) $t = 0$, when the two Kirks are started at the same time. (b) $t = 16$, when the astronaut observed that the science experiment was setup. (c) $t = 23$, when the robot received a delayed observation from the astronaut indicating they had completed science setup. (d) $t > 23$, as the robot performed the drilling task.

Chapter 5 expands existing algorithms for deciding when to act given the resolution of constraints. There, we contribute a novel strategy for deciding when to act given observation delay. In Chapter 6, we formalize the components of task scheduling executive that can be deployed to real hardware. Chapter 7 finally contributes a multi-agent coordination architecture for environments with uncertain communication. The discussion in Chapter 8 concludes this thesis by providing additional context for the decisions made during this research.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Problem Statement

There is a real-world need for coordinating multiple agents that are collaborating while facing uncertain observations and uncertain inter-agent communication. We focus on an example drawn from the domain of human spaceflight below, but any domain in which operations take place in extreme environments needs to consider uncertain communication. Examples include military operations and remote science, both summarized below.

For instance, military operators are frequently in environments where communication is impossible or dangerous. Radioing from a staging area back to a command center may be impossible. Even communications between operators in a hostile environment may risk exposing them to an enemy. Despite communication concerns, there is an advantage for operations who can coordinate on a battlefield. A delay scheduler would allow military operators to decide when to act given uncertainty in their environment and communications.

Remote science with robotic explorers features many of the same communication and observation constraints that are found in human spaceflight. The upcoming set of NASA Commercial Lunar Payload Systems (CLPS) [18] rovers will be collecting samples and exploring the lunar surface. While a CLPS rover is a single agent in and of itself, it is important to recognize that both flight controllers and the scientists on the ground need to pass information between each other and the rover to make the mission successful. From a scheduling perspective, CLPS mission execution is

a multi-agent scenario. The scientists and flight controllers running CLPS missions must contend with uncertain or delayed communications. For example, there will likely be planned and unplanned loss of signal events, uncertain delays in the arrival of scientific information due to protocols like CFDP [19], and issues with topological interference on the lunar surface.

Consider a subset of activities a rover might perform in a science station, or a small area where multiple samples are collected. Within a tightly constrained window at each science station (measured on the order of hours), a rover may collect core samples from multiple sites. Drill sites within the science station might be pre-planned, however, it is possible that some later sites can be changed by scientists based on the data collected so far. If that is the case, scientists would have a limited time window after the first samples arrive to affect where later samples are collected. A delay scheduler would allow the scientists to identify the last possible moment to suggest a new drill site for the end of the science station, hence giving them as much time as possible to analyze the data they receive and debate what other site might be most valuable. Without taking communication delay into account, scientists may debate too long and miss an opportunity to impact drill sites.

We now provide an example using EVAs to motivate the need for uncertain observations.

2.1 EVAs as a Problem Domain

EVAs must be performed in a timely manner. Life support imposes an upper bound on the total duration of an EVA. As an EVA progresses, EV crews consume four non-renewable resources, comprised of oxygen, battery power, water, and CO_2 scrubbers [20]. The duration of EVAs is limited by the consumable that is on track to be depleted first across the life support systems of both EV crew members, referred to as the limiting consumable. Absent any other more pressing constraint, it is this limiting consumable that forces EVA crews and ground support to stay on timeline.

Meanwhile, uncertain observations are manifested across three distinct categories:

signal transmission, human operational delays, and instrument processing. Each category presents a source of delay that varies with uncertainty. For signal transmission, delay is sourced from the speed of light between planetary bodies and infrastructural deficiencies. Communication infrastructure outside of low-Earth orbit, including satellites and planetary surface signal repeaters [21], is not robust, and as such unpredictable delays and signal dropouts will be common [22]. For human operational delays, note that the primary goal of Mission Control is keeping the crew safe [23]. As such, communications from the science team to the crew may be delayed or dropped because Mission Control needs to prioritize communications and actions related to crew health and safety at the expense of science [24], [25]. Lastly, for instrument processing, there is uncertainty in the temporal relationships between the activation of complex scientific instruments and the return of useful information [26]. Scientific packages may generate high and low bandwidth data products, the uplink of which will be bottlenecked by limited bandwidth between space and ground.

Consider a spacewalker who is installing an array of satellite dishes on the Moon. The procedure for installing a single satellite dish is well defined. The procedure involves, say, firmly inserting a tripod into the lunar regolith, putting a dish on top, bolting the dish in, attaching a few wires, and waiting to get confirmation from ground that the dish is operational. Sometimes the tripod is easy to burrow into the regolith, other times it takes a few tries. Sometimes the confirmation comes quickly, other times it takes time. Some dishes are to be placed close to one another, yet others should be far apart and across difficult to traverse terrain. Once one dish is done, the astronaut can move on to the next. All the while, the astronaut's life support system is slowly draining its consumable resources (e.g. oxygen and battery). Ground wants one dish tested at a time, so the astronaut must wait for the confirmation before proceeding. But the astronaut also knows the confirmation will come *eventually*. They can continue to the next dish before receiving the confirmation if they are confident that doing so still guarantees that the next installation will not happen before the ground is ready. Another way of looking at it is that the astronaut knows ground confirmed the installation, but the communication saying as much was

delayed. The challenge then is to wait as little time as necessary before moving on to the next dish. To decide when to act, the astronaut relies on advice from the delay scheduler built in to their digital assistant.

2.2 Problem Statement Definitions

Broadly speaking in this scenario, we have two types of input and one type of output. The first input is decided before the astronaut egresses the habitat. MCC writes an EVA timeline (much like Figure 1-3), which describes the events that need to take place and the relationship between events, such as their ordering, the time between them, or how much delay there may be between when an event occurs and when it is observed. Once the astronaut starts the EVA, we have a second input, which is the time when events are observed. Taken together, we are tasked with finding an output of deciding which future events should be executed at what time.

We use temporal networks [9] to model EVA timelines as temporal constraints between a finite set of events. Some events may have associated uncertain observation delay. Let a temporal network be represented by S, which is a tuple of events X and constraints R, $\langle X, R \rangle$. Constraints take the form of set-bounded intervals between two events. Some events in a temporal network may be associated with an uncertain observation delay $\bar{\gamma}$.

At some time t during an EVA, we have a set of events that were *observed* before t, $\texttt{obs}(x < t)$. When an event has been recorded at a given time t, we say that it has been *assigned*. Both observations and assignments are mappings from an event to a time in $\mathbb{R}$.

The set of events that were assigned a time before t is $\xi(x < t)$. If there is no associated observation delay with an event x_c, then $\texttt{obs}(x_c) = \xi(x_c)$. If there is associated observation delay, then it is possible that $\texttt{obs}(x_c) < \xi(x_c)$.

We want a *Real-Time Execution Decision* (RTED), which consists of unexecuted events and when they should be performed. Each RTED is a tuple of a set of unexecuted events, $x_u \subseteq X$ and future time, t', $\langle x_u, t' \rangle$.

Our specific problem statement for the delay scheduler is as follows.

Definition 1. Single-Agent Delay Scheduler

The delay scheduler should take triple $\langle S, \bar{\gamma}, \texttt{obs}(x < t)\rangle$ of the offline (before scheduling) and online (during scheduling) components of scheduling as input. It must output an RTED $\langle x_u, t'\rangle$.

We can expand the scenario from above to include multiple astronauts installing multiple satellite dishes in parallel. MCC wants to minimize the number of dishes that are being confirmed at any given moment. We add new *inter-agent* constraints dictating that, given astronauts 1 and 2, astronaut 2 may not start installing a dish until they receive confirmation that astronaut 1 is complete. Likewise, astronaut 3 must wait for 2 to finish their confirmation, 4 must wait for 3, and so on in a round robin fashion. Like communication with MCC, communications between astronauts is spotty (hence why they need to install communication infrastructure!) Sometimes, astronauts may easily communicate, other times, communications may be significantly delayed or drop out altogether. Naturally, the astronauts must be able to share events with each other to satisfy the inter-agent constraints.

We expand the previous problem statement to the multi-agent case by adding the notion of agents, A, each with their own delay scheduler. Each delay scheduler has their own S_a with a subset of events, $x \subset X$, they expect to receive from their peers in the form of observations or communications. While some actions are aligned between agents, there is no assumption that all agents are working against the same events with the same constraints. From the perspective of an agent, $a \in A$, at time t, their peers simply need to be aware of what events a has assigned up to t, $\xi_a(x \leq t)$. Events that the peers of a communicate to a are no different than observations of the environment that a makes.

We must define a problem statement for how delay schedulers should coordinate in a multi-agent context.

Definition 2. Multi-Agent Event Communications

Given online input of tuple $\langle \xi(x_t), A$, agent a should output all assignments $\xi_a(x_t)$ in the form of a broadcast to all other agents, $A - \{a\}$.

In other words, event assignments should be broadcasted to all peers as soon as an assignment is made.

Chapter 3

Approach

We define a delay scheduler in such a way that one instance is useful for single-agent scheduling, and multiple instances can be seamlessly integrated for collaboration with inter-agent constraints. Our approach builds towards such a multi-agent delay scheduler by first defining the subproblems necessary for single-agent scheduling with uncertain communication before layering on a communication pathway for multi-agent scheduling. Figure 3-1 presents a simplified view of multi-agent scheduling.

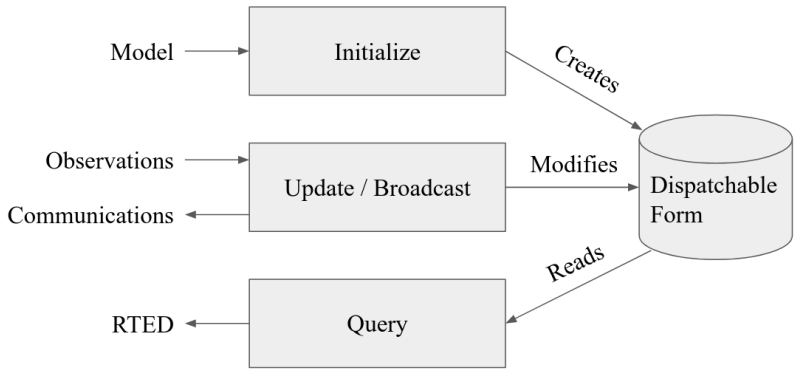


Figure 3-1: A sample architecture with two delay schedulers collaborating. Each agent receives a single temporal network as input. Observations of the outside world are recorded. Communications relay event assignments to peers. Each agent outputs its own RTED.

Our approach relies on the ability of a delay scheduler to accurately decide what events should be scheduled and when it is *valid* to do so. In this context, a decision being "valid" means that the events and time of an RTED guarantees that all con-

straints in the problem can be satisfied given the history of events scheduled before now. We naturally need one or more data structures that, if maintained correctly during scheduling, can be queried to produce such an RTED. We refer to such a data structure as the dispatchable form, though, as will be seen in Chapter 5, other data structures facilitate scheduling as well.

A delay scheduler must be able to output RTEDs that are consistent with the constraints of the problem and the history of scheduled events. There are four key subproblems that must be addressed:

1. an offline process must **initialize** a dispatchable form that reflects the semantics of S and $\bar{\gamma}$,
2. an online process must **update** the dispatchable form given an event observation at a given time,
3. an online process must **broadcast** new event assignments with peers, and
4. an online process must **query** the dispatchable form for new RTEDs.

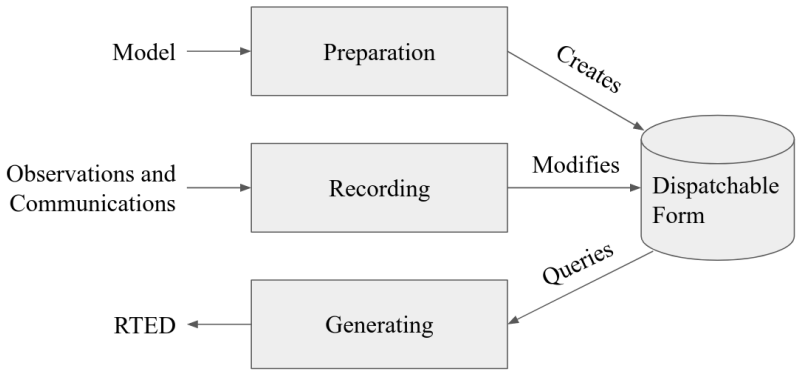


Figure 3-2: The four interfaces of a delay scheduler. The second and third are combined to highlight that broadcasts are trigged when events are observed. The first shows the dispatchable form being intialized from a model. The second shows that event observations will cause the dispatchable form to be updated, immediatelly triggering a broadcast (the third interface). The fourth interface queries the dispatchable form to create RTEDs.

Figure 3-2 shows the architecture of a delay scheduler with respect to its four interfaces and dispatchable form.

Input: Controllable temporal network S; Observation uncertainty $\bar{\gamma}$; `clock`; `peers`
Initialization: `dispatchable-form` $\leftarrow$ `initialize`$(S, \bar{\gamma})$; `RTED` $\leftarrow \varnothing$;
Delay Scheduling:

```
1  while there are unexecuted executable events do
2  |   if Event x is observed then
3  |   |   update(dispatchable-form, x, clock.now));
4  |   |   broadcast(x, peers);
5  |   endif
6  |   RTED ← query(dispatchable-form, clock.now)
7  end
```

Algorithm 1: Algorithm for performing delay scheduling to produce RTEDs for all executable events in a temporal network.

We provide pseudo-code for a delay scheduler in Algorithm 1. `initialize` will create the dispatchable form, `update` will modify the dispatchable form to reflect an event assignment, `broadcast` will send event assignments to peers, and `query` will read the dispatchable form to find the next RTED.

For now, we make the following assumptions. We assume that initialization, updates, and queries are sound and complete algorithms with respect to their intended handling of the dispatchable form. We use the term "networked" to refer to agents that can communicate event observations to each other. Broadcasting assumes the existence of a communication protocol that guarantees messages indicating an event has been assigned reach all networked schedulers. If a temporal network is controllable, then there must exist a dispatchable form [27].

Below, we prove that Algorithm 1 will guarantee the output of valid RTEDs for all executable events for coordinating agents. We start by showing that the delay scheduler can be used in a multi-agent context.

Lemma 1. *For a delay schedulers, if each S_a and $\bar{\gamma}_a$ received by each delay scheduler $a \in A$ is controllable and accurately models the world, then all networked delay schedulers may produce valid RTEDs.*

Proof. If S_a is controllable for a single delay scheduler, then there must exist a set of RTEDs that allows all constraints to be satisfied for all resolutions of uncertainty in the uncontrollable constraints and observation delay. If it were not the case that any

S_a and $\bar{\gamma}_a$ accurately models the world, then the uncontrollable inter-agent constraints of S_a and $\bar{\gamma}_a$ would not strictly encompass all possible outcomes of uncertainty during scheduling. This would mean that, during scheduling, there may be a resolution of uncertainty that does not allow a valid RTED to be produced, which is inconsistent with a controllable S_a. Thus if each S is controllable and accurate for all delay schedulers, then all delay schedulers may produce valid RTEDs. □

We now show that the delay scheduler will produce valid RTEDs in a single-agent context.

Lemma 2. *Given a controllable temporal network S consisting of a set of events, X, constraints, R, and uncertain observation delay, $\bar{\gamma}$, initializing the dispatchable form before the first event is observed guarantees the dispatchable form can be used to schedule any executable event in X.*

Proof. As an offline process, by definition initialization will run before scheduling begins. Thus, the dispatchable form must be valid and include enough information to schedule all executable events. □

Not all events may be observed. For instance, any event with infinite observation delay (as may occur during a communication dropout) is unobservable. The delay scheduler cannot schedule events with infinite observation delay.

Lemma 3. *All events that have the ability to impact RTEDs may be observed by the delay scheduler.*

Proof. If S is controllable, then it must be the case that the delay scheduler can output RTEDs that will guarantee all constraints between events can be satisfied. If it were the case that unobservable events must be assigned in order to produce a valid RTED, then scheduling would depend on information that cannot be learned, meaning S would not be controllable. □

For the next lemma, it is important to highlight that RTEDs may not depend on information about future events.

Lemma 4. *If an event is scheduled, it will be observed by the delay scheduler.*

Proof. There are two parts to this proof. First, the delay scheduler loops without pause until all executable events have been scheduled, meaning that the conditional on line 2 will be reached for all events that can have an impact on RTEDs.

Second, RTEDs are not the same as event assignments. As will be shown in Definition 29, the execution time of an RTED must be in the future. We are not allowed to assign an event to a future time, thus the events in an RTED cannot be immediately scheduled. We observe when executable events are scheduled in the future. □

Lemma 5. *If we observe all events before producing RTEDs, then RTEDs will always be valid.*

Proof. We see that we always check for event observations before producing RTEDs. There are no processes between checking for an observation and producing an RTED. Therefore, each RTED will be queried against a dispatchable form that has been modified to reflect all event assignments up to the current time. If the choice of dispatchable form is valid for any set of assignments up to the current time, and the querying process is sound and complete, then the RTED must also be valid. □

We finish by revisiting the multi-agent context.

Lemma 6. *If there are satisfiable inter-agent constraints, then broadcasting all event assignments to all peers guarantees that each delay scheduler may produce valid RTEDs.*

Proof. All observable events must be assigned. If all event assignments are broadcasted to all agents, then it must be the case that all agents observe all events. If all observable events are received for a controllable S, then it must be the case that a delay scheduler can produce a valid RTED, and thus all networked delay schedulers can produce valid RTEDs. □

The next chapters will address each of the assumptions made above. Chapter 4 will elaborate on modeling temporal networks with uncertain communication and checking

their controllability. Chapter 5 will focus on the process of creating and maintaining a dispatchable form throughout single-agent scheduling. Chapter 6 describes the integration of Algorithm 1 in a high-level task executive. Chapter 7 will describe the design of a robust broadcasting algorithm for networked schedulers with uncertain communication.

Chapter 4

Modeling Temporal Constraints with Uncertain Observation Delay

Our overarching aim in this Chapter is to architect the offline portions of a single-agent pipeline that takes a temporal network with uncertain observation delay of exogenous events as input in order to execute events in the real world within time windows that guarantee temporal consistency. Given everything we know and do not know about the temporal relationship between events within and outside of our control, this chapter will lay the groundwork that there exists an execution strategy that guarantees all constraints are satisfied despite the uncertainty. The next chapter, Chapter 5, will provide accompanying online procedures for acting on said execution strategy and dispatching events with real hardware (or by generally telling an agent what to do).

The three aims of this chapter are to

1. model temporal constraints with uncertain observation delay,
2. define a consistency (controllability) checking procedure for temporal networks with uncertain observation delay, and
3. prove that the execution strategy assumed by the controllability checking procedure is safe.

In Section 4.1, we address aim (1) by outlining the necessary definitions for model-

ing temporal networks. Sections 4.2 and 4.3 describe our chosen model for temporal constraints with uncertain observation delay, and address aim (2) by presenting a procedure for checking the consistency thereof. Sections 4.2 and 4.3 are largely based on the work originally put forth by Bhargava et. al., [13], [28], [29] with additional contributions by us as highlighted below. Notably, Section 4.3 addresses aim (3) by contributing novel proofs that the execution strategy assumed to exist by Bhargava et. al. is safe. We conclude with experimental analysis of our chosen consistency checking procedure in Section 4.5 using example temporal constraint networks inspired by lunar exploration.

4.1 Temporal Networks

Temporal networks form the backbone of our architecture for temporal reasoning under observation delay. Simple Temporal Networks (STNs) offer the basic building blocks for most expressive temporal network formalisms [9]. An STN is composed of a set of variables and a set of binary constraints, each of which limits the difference between a pair of these variables; for example, $B - A \in [10, 20]$. Each variable denotes a distinguished point in time, called an *event*. Constraints over events are binary *temporal constraints* that limit their temporal difference; for example, the aforementioned constraint specifies that event A must happen between 10 and 20 minutes before event B.

Definition 3. STN [9]

An *STN* is a pair $\langle X, R \rangle$, where:

- X is a set of variables, called events, each with a domain of the reals $\mathbb{R}$, and
- R is a set of simple temporal constraints. Each constraint $\langle x_r, y_r, l_r, u_r \rangle$ has scope $\{x_r, y_r\} \subseteq X$ and relation $x_r - y_r \in [l_r, u_r]$.

Definition 4. Schedule [27]

A *schedule*, ξ, is a mapping of events to times, $\xi : X \to \mathbb{R}$.

An STN is used to frame scheduling problems. A schedule is feasible if it satisfies each constraint in R. We use the notation $\xi(x)$ to represent a mapping from an event, x, to a time, $x \to \mathbb{R}$, in the schedule. A schedule is *complete* if all $x \in X$ are assigned times in ξ. An STN is *consistent* if it has at least one feasible schedule that assigns all events in X.

An STN is consistent if and only if there is no negative cycle in its equivalent distance graph [9]. Let n be the number of events in a temporal network and m to be the number of constraints. Then consistency of an STN can be checked in $O(mn)$ time using the Bellman-Ford Algorithm [30] to check for negative cycles.

While an STN is useful for modeling problems in which an agent can control the exact time of all events, it does not let us model actions whose durations are uncertain. A Simple Temporal Network with Uncertainty (STNU) is an extension to an STN that allows us to model these types of uncertain actions [10].

Definition 5. STNU [10]

An *STNU* S is a quadruple $\langle X_e, X_c, R_r, R_c \rangle$, where:

- X_e is the set of executable events with domain $\mathbb{R}$,
- X_c is the set of contingent events with domain $\mathbb{R}$,
- 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 $l_r, u_r \in \mathbb{R}$, and
- 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 $l_r, u_r \in \mathbb{R}$.

An STNU divides its events into executable and contingent events and divides its constraints into requirement and contingent constraints. The times of executable events are under the control of an agent, and assigned by its scheduler. STNU executable events are equivalent to events in an STN. Contingent events are controlled by Nature. Contingent constraints model the temporal outcomes of uncertain actions and are enforced by Nature. Contingent constraints relate a starting executable event and an ending contingent event. To ensure causality, the lower-bound of a contingent constraint is required to be non-negative; hence, the end event of the constraint

follows its start event. Contingent constraints are not allowed to be immediately followed by additional contingent constraints. Requirement constraints specify constraints that the scheduler needs to satisfy and may relate any pair of events. An STNU requirement constraint is equivalent to an STN constraint.

To clarify terminology, we sometimes refer to contingent constraints as contingent links and requirement constraints as requirement links. We also sometimes use the term *free* instead of requirement to describe executable events and constraints. When we discuss contingent constraint or contingent link duration, we refer to the amount of time that actually elapses between a contingent link's starting executable event and its ending contingent event. We sometimes refer to STNUs as defined in Defintion 5 as *vanilla* STNUs (in contrast to the many "flavors" of STNUs, namely the variants with fixed and variable observation delay functions as will be defined in Sections 4.2 and 4.3 respectively).

With STNs, our goal is to construct a consistent schedule for all events such that all constraints are satisfied. In STNUs, however, contingent events cannot be scheduled directly. Instead, we are interested in determining whether there is a *controllable* execution strategy that guarantees that a schedule can be constructed such that all constraints are satisfied despite how uncertainty is resolved.

Definition 6. Situations [10]

For an STNU S with k contingent constraints $\langle e_1, c_1, l_1, u_1\rangle, \cdots, \langle e_k, c_k, l_k, u_k\rangle$, each *situation*, ω, represents a possible set of values for all links in S, $\omega = (\omega_1, \cdots, \omega_k) \in \Omega$. The *space of situations* for S, Ω, is $\Omega = [e_1, c_1] \times \cdots \times [e_k, c_k]$.

Each *situation* in the *space of situations*, $\omega \in \Omega$, represents a different assignment of contingent links in the schedule [10]. We may represent the situation for a specific constraint as ω_i for the i-th constraint in S, or $\omega(x_c)$ for contingent event x_c.

Situations may be applied to STNUs.

Definition 7. Projection [10], [27]

A *Projection* is an application of a situation, ω, on an STNU S, which collapses the durations of contingent links to specific durations resulting in an STN.

A *projection* is an STN that is the result of applying a situation to an STNU, and thus the contingent links have reduced from uncertain ranges to specific durations [10], [27].

Definition 8. Execution Strategy

An *execution strategy*, $\mathcal{S}$, is a mapping of situations to schedules, $\mathcal{S} : \Omega \rightarrow \Xi$.

An *execution strategy* then naturally maps a specific resolution of the uncertainty of the contingent constraints to a set of assignments for the events of an STNU. For an STNU, time monotonically increases and we only observe *activated* contingent events, or those contingent events at the tail of a contingent link whose free event predecessor has been executed. As such, we modify our definition of ξ.

Definition 9. Partial Schedule

A *partial schedule*, ξ, is a mapping from a proper subset of events in an STNU, $X' \subseteq X_e \cup X_c$, to times, $\xi : X' \rightarrow \mathbb{R}$.

As a proper subset, ξ represents an assignment of events *so far* during the execution of an STNU. From here on, ξ refers to a partial schedule. If $X' = X_e \cup_c$, then the schedule is complete.

To determine whether an STNU is controllable, we determine whether there exists a *valid* execution strategy for it.

Definition 10. Valid Execution Strategy

A *valid* $\mathcal{S}$ is one that enforces that, for any $\omega_f \in \Omega_f$, the outputted decision respects all existing temporal constraints and ensures the existence of a subsequent valid execution strategy following that action.

In the world of STNU literature, there are many forms of controllability that represent the ability of a scheduler to enact execution strategies that satisfy constraints under different conditions [10]. Three forms of controllability, *strong*, *weak*, and *dynamic* are studied most often, though in practice we omit weak controllability from our analysis. A temporal network is *strongly controllable* (or exhibits strong controllability) (SC), if there exists a complete schedule that will satisfy all constraints for

all projections of the STNU. A temporal network exhibits dynamic controllability (DC) if an execution strategy exists for a given partial schedule. As we will see below, variable-delay controllability, used to check the consistency of temporal networks with uncertain observation delay, will unify strong and dynamic controllability into a single theory. But first, we describe fixed-delay controllability, which introduces known observation delay to STNUs.

4.2 Fixed-Delay Controllability

Under fixed-delay controllability (FDC) [28], we consider the problem of scheduling execution decisions when the assignment of values to contingent events is learned after some time has passed from the initial assignment, if ever. Fixed-delay controllability uses a *fixed-delay function* to encode the delay between when an event occurs and when it is observed by a scheduling agent. We sometimes refer to an STNU with an associated fixed-delay function as a *fixed-delay STNU*.

Definition 11. Fixed-Delay Function [28]

A *fixed-delay function*, $\gamma : X_c \rightarrow \mathbb{R}^+ \cup \{\infty\}$, maps a contingent event to the amount of time that passes between when the event is assigned and when its value is observed.

As a matter of convention, we use $A \xrightarrow{[l,u]} B$ to represent requirement links between events A and B and use $A \xRightarrow{[l,u]} E$ to represent contingent links between A and E. When we refer to the fixed-delay function associated with a contingent event E of some contingent constraint $A \xRightarrow{[l,u]} E$, we use the notation $\gamma(E)$, or equivalently, γ_E. Without instantaneous observation of contingent events, we must clarify the relationship between when an event is assigned and when it is *observed*.

Definition 12. Contingent Event Observation

Observations, `obs`, are a mapping from contingent events to times when the agent receives knowledge the event has been assigned, $\texttt{obs} : X_c \rightarrow \mathbb{R}$. An observation of an event, x_c, follows the relationship, $\texttt{obs}(x_c) = \xi(x_c) + \gamma(x_c)$.

We also present a revised definition of situations, Ω_f, to reflect the impact of the delay function on event observations.

Definition 13. Fixed-Delay Situations

For an STNU S with k contingent constraints $\langle e_1, c_1, l_1, u_1\rangle, \cdots, \langle e_k, c_k, l_k, u_k\rangle$ and fixed-delay function γ, each *fixed-delay situation*, ω_f, represents a possible set of *observed* values for all links in S, $\omega_f = (\omega_{f1}, \cdots, \omega_{fk})$. The {space of situations} for S, Ω_f, is $\Omega_f = [e_1, c_1] + [\gamma_1, \gamma_1] \times \cdots \times [e_k, c_k] + [\gamma_k, \gamma_k]$.

To emphasize that the *observed* value for an event is not the same as its assignment, we also use the term *observation space* as a synonym for the space of situations.

Definition 14. Valid, Fixed-Delay Execution Strategy

A *valid* $\mathcal{S}$ for a fixed-delay STNU is one that enforces that, for any $\omega_f \in \Omega_f$, while receiving observations of contingent events after a known and fixed delay, the outputted decision respects all existing temporal constraints and ensures the existence of a subsequent valid execution strategy following that action.

With the semantics of delayed observations in hand, we can define what it means for a fixed-delay STNU to be controllable.

Definition 15. Fixed-Delay Controllability [28]

An STNU S is *fixed-delay controllable* with respect to a delay function, γ, if and only if for the space of situations, Ω_f, there exists a valid, fixed-delay execution strategy, $\mathcal{S}$, that will construct a satisfying schedule for all requirement constraints during execution.

Importantly, fixed-delay controllability (FDC) generalizes the two concepts of controllability that are central to STNUs, strong and dynamic controllability. In particular, by using a fixed-delay function where we observe all events instantaneously, e.g. $\gamma(x_c) = 0 \forall x_c \in X_c$, checking fixed-delay controllability reduces to checking *dynamic controllability*. Similarly, a fixed-delay function that specifies we never observe any contingent events, e.g. $\gamma(x_c) = \infty \forall x_c \in X_c$, corresponds to checking *strong controllability* [10].

As is the case for a vanilla STNU, evaluating whether a valid execution strategy exists for a fixed-delay STNU reduces to checking for the presence of a *semi-reducible negative cycle* in a *labeled distance graph* derived from the fixed-delay STNU [31]. The key insight for checking fixed-delay controllability is the inclusion of a fixed-delay function in the constraint generation rules for building the labeled distance graph [28].

The labeled distance graph corresponds to the constraints of the STNU with each unlabeled edge from A to B with weight w (denoted $A \xrightarrow{w} B$) representing the inequality $B - A \leq w$. Labeled edges represent conditional constraints that apply depending on the realized value of contingent links in the graph. For example, a lower-case labeled edge from A to B with weight w and lower-case label c (denoted $A \xrightarrow{c:w} B$) indicates that $B - A \leq w$ whenever the contingent link ending at C takes on its lowest possible value. An upper-case labeled edge from A to B with weight w and upper-case label C (denoted $A \xrightarrow{C:w} B$) indicates that $B - A \leq w$ whenever the contingent link ending at C takes on its highest possible value. Given a labeled distance graph, there are several valid derivations we can apply to generate additional edges (see Table 4.2). If it is possible to derive a negative cycle that is free of lower-case edges, then the STNU has a *semi-reducible negative cycle* and the STNU is not controllable.

Note that with fixed-delay controllability, the lower-case and cross-case rules are modified from the Morris and Muscettola [32], accounting for γ. More specifically, we address the case where observation delay makes it impossible to receive information about a contingent event before its immediate successor. More detail can be found in [33].

We generalize fixed-delay to variable-delay controllability next.

4.3 Variable-Delay Controllability

While fixed-delay controllability is quite expressive, its fundamental limitation is that it assumes that contingent event assignments, even those made after a fixed delay, are

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

Table 4.1: Edge generation rules for a labeled distance graph derived from a fixed-delay STNU.

always known. If uncertainty in observation delay, and thus uncertainty in contingent event assignment, is added to the model, then we are forced to decide when to act despite imperfect knowledge of the partial history.

We now introduce this model in terms of definitions for a *variable-delay function* and *variable-delay controllability* (VDC) checking as applied to *variable-delay STNUs*. Since variable-delay semantics generalizes the notion of fixed-delay, as a matter of convenience, we also use the simplified term *delay STNUs* to refer to STNUs with variable observation delay. VDC was originally presented by Nikhil Bhargava [13]. However, we contributed significant improvements of the lemmas and proofs herein, including the addition of novel visual depictions of VDC, in our role as a coauthor with Bhargava on a journal article on the topic of VDC that was submitted to the *Journal of AI Research*.

This section formalizes the definition of VDC, which is required to explain the procedure of checking VDC in Section 4.3.1.

Definition 16. Variable-Delay Function

A *variable-delay function*, $\bar{\gamma} : X_c \rightarrow (\mathbb{R}^+ \cup \{\infty\}) \times (\mathbb{R}^+ \cup \{\infty\})$, maps a contingent event, x_c, to an interval $[a, b]$, where $a \leq b$. The interval bounds the time that passes after $\xi(x_c)$ before that value is observed to be assigned. No prior knowledge is assumed about the distribution associated with this interval.

Importantly, this model does not assume that an executing agent may be able to infer *when* a contingent event was executed. Instead, our model only infers *that*

the event was executed. Like the resolution of contingent constraints, the resolved value of $\bar{\gamma}(x_c)$ will be selected by Nature during execution. Thus, the timing of when an agent receives an observation is a function of the independent resolutions of the contingent link and variable-delay function.

By convention, we use $\bar{\gamma}^-(x_c)$ and $\bar{\gamma}^+(x_c)$ to represent the lower-bound and upper-bound, respectively, of the range representing the possible delay in observation, i.e. $\bar{\gamma}(x_c) \in [\bar{\gamma}^-(x_c), \bar{\gamma}^+(x_c)]$.

Definition 17. Observation Projection

The *observation projection* Γ is a mapping from a contingent event to a fixed observation delay, $\Gamma : X_c \to \mathbb{R} \in [\bar{\gamma}^-(X_c), \bar{\gamma}^+(X_c)]$.

During execution, the *observation projection*, Γ, represents the resolution of observation delay. Much like how a projection collapses a vanilla STNU to an STN, the observed projection collapses a contingent link with variable-observation delay to one with fixed-observation delay. However, unlike the projection of an STNU, the observation projection is not guaranteed to be learned. We update our definitions of $\texttt{obs}$, ξ, and Ω accordingly.

Definition 18. Contingent Event Observation

Contingent event observations, $\texttt{obs}$, are a mapping from contingent events to times when the agent receives events, $\texttt{obs} : X_c \to \mathbb{R}$, based on the relationship, $\texttt{obs}(x_c) = \xi(x_c) + \Gamma(x_c)$.

Determining a real-valued mapping of a contingent event to the value of its assignment, i.e. its schedule or $\xi(x_c)$, is no longer guaranteed due to an interval bounded $\Gamma(x_c)$. We must use interval-bounded contingent event assignments instead.

Definition 19. Schedule

A *schedule*, ξ, when applied to contingent events, is a mapping of events to interval-bounded times, $\xi : X_c \to (\mathbb{R}^+ \cup \{\infty\}) \times (\mathbb{R}^{++} \cup \{\infty\})$, where, for any contingent constraint, $0 \leq l_r \leq c_r - e_r \leq u_r$, ending in contingent event x_c, $\xi(x_c) \in [l + \bar{\gamma}^-(x_c), u + \bar{\gamma}^+(x_c)]$.

We sometimes use interval bounded schedules for requirement events as well. For a requirement constraint $l_r \leq x_r - y_r \leq u_r$ ending in requirement event x_e, $\xi(x_e) = t \in [l_r, u_r]$ for some time t.

We once again revise our definition of situations, Ω_v, to reflect the impact of the variable-delay function on the space of observations.

Definition 20. Variable-Delay Situations

For an STNU S with k contingent constraints $\langle e_1, c_1, l_1, u_1\rangle, \cdots, \langle e_k, c_k, l_k, u_k\rangle$ and variable-delay function $\bar{\gamma}$, each *variable-delay situation*, ω_v, represents a possible set of *observed* values for all links in S, $\omega = (\omega_{v1}, \cdots, \omega_{vk})$. The {space of situations} for S, Ω_v, is $\Omega_v = [e_1, c_1] + [\bar{\gamma}_1^-, \bar{\gamma}_1^+] \times \cdots \times [e_k, c_k] + [\bar{\gamma}_k^-, \bar{\gamma}_k^+]$.

We see that the space of observations has likewise grown in the transition to variable observation delay. If $\bar{\gamma}^- < \bar{\gamma}^+$, Ω_v for variable observation delay is strictly larger than Ω_f for fixed-observation delay and Ω for vanilla STNUs.

Like the fixed-delay function for fixed-delay controllability, the variable-delay function relates an observation delay to a contingent event, independent of other events. We take a similar approach to defining variable-delay controllability, relative to fixed-delay controllability.

Definition 21. Variable-Delay Controllability

An STNU S is *variable-delay controllable* with respect to a variable-delay function, $\bar{\gamma}$, if and only if for the space of situations, Ω_v, there is an $\mathcal{S}$ that produces a satisfying schedule for requirement events during execution, ξ.

Determining whether a given variable-delay STNU, S, is variable-delay controllable has two components [13]. The first is to derive a fixed-delay STNU, S', with fixed-observation delay, γ, that is equivalent with respect to controllability. The second is to show that S' is fixed-delay controllable. Below, we reiterate the claims of [13], demonstrating how to derive S' from S that is equivalent with respect to controllability. In Section 4.3.1, we first demonstrate how to transform the contingent links from S to S', and demonstrate their correctness with respect to observation

spaces, before following up with transformations to the requirement links to maintain the same scheduling semantics in S'.

4.3.1 Variable-Delay to Fixed-Delay Transformations

We now show how we transform a variable-delay STNU to a fixed-delay STNU in order to perform fixed-delay controllability checking.

For the following lemmas, let x_c be a contingent event in S and variable-delay function $\bar{\gamma}(x_c)$. Let x'_c be the transformed contingent event in S' with fixed-delay function, $\gamma(x'_c)$.

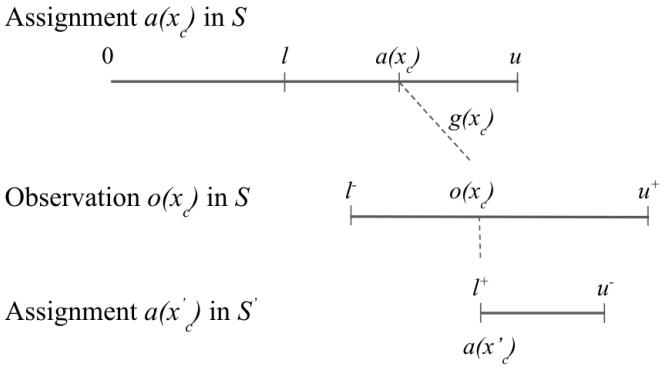


Figure 4-1: We visualize the relationship between realized assignments across S and S'. In this example, each horizontal line is a timeline monotonically increasing from left to right. Dashed lines represent observation delays. We see how an assignment in S, $\xi(x_c)$, realized observation delay, $\Gamma(x_c)$, and an observation in S, $\texttt{obs}(x_c)$, contribute to an assignment in S', $\xi(x'_c)$.

Note that we receive $\texttt{obs}(x_c)$ from Nature, but make the assignment $\xi(x'_c)$ in the dispatchable form of S'. To be clear, while $\xi(x_c)$ is an interval, $(\mathbb{R} \cup \infty) \times (\mathbb{R} \cup \infty)$, $\xi(x'_c)$ is in $\mathbb{R}$. For a fixed interval, e.g. $\texttt{obs}(x_c) \in [t, t]$, we sometimes employ an equivalent representation, $\xi(x_c) = t$.

Additionally, we sometimes apply $-$ and $+$ superscripts to l and u to denote the earliest and latest times respectively that an assignment at those bounds could be observed. For instance, the relationship in Definition 18 simplifies to,

$$\texttt{obs}(x_c) = [l + \bar{\gamma}^-(x_c), u + \bar{\gamma}^+(x_c)] \tag{4.1}$$

$$\texttt{obs}(x_c) = [l^-(x_c), u^+(x_c)] \tag{4.2}$$

Lastly, we need a means to compare observation spaces if we are to transform variable-delay to fixed-delay STNUs.

Definition 22. Observation Space Mapping

Let μ be a mapping from an assignment to a situation, $\mu : \xi \rightarrow \omega$. To say that $\mu(x'_c) \subseteq \omega_v(x_c)$ means that, for any assignment of x'_c in S', there is an equivalent situation in S for x_c.

For the transitions below, it is a *valid observation space mapping*, if we can show that $\mu(x'_c) \subseteq \omega_v(x_c)$. If so, it is guaranteed that any assignment in the observation space of x'_c also has a valid assignment in the observation space of x_c.

We now have the necessary vocabulary and notation to step through the transformations from S to S'. These lemmas were first presented in [13], with some refinement by us for the aforementioned journal article submission.

Definition 23. Variable-Delay to Fixed-Delay Transformations

The *variable-delay to fixed-delay transformations* define a set of observation space mappings, where there are valid observation space mappings for all the contingent constraints in S' to S.

Thus, if there is a satisfying $\mathcal{S}$ for the fixed-delay observation space of S', it is guaranteed to simultaneously satisfy any situation in the variable-delay observation space, Ω_v, of S.

Lemma 7. *For any contingent event $x_c \in X_c$ in S, if $\bar{\gamma}^-(x_c) = \bar{\gamma}^+(x_c)$, we emulate $\bar{\gamma}(x_c)$ in S' using $\gamma(x'_c) = \bar{\gamma}^+(x_c)$.*

Proof. We translate an already fixed-bounded observation delay in the form of $\bar{\gamma}(x_c)$ to the equivalent fixed-delay function, $\gamma(x'_c)$, thus $\omega_f(x'_c) = \omega_v(x_c)$. □

Lemma 8. *For any contingent event $x_c \in X_c$, $\bar{\gamma}^+(x_c) = \infty$, we emulate $\bar{\gamma}(x_c)$ in S' as $\gamma(x'_c) = \infty$.*

Proof. There are projections where we would not receive information about x_c, therefore we have to act as if we *never* receive an observation of x_c. Any $\mathcal{S}$ that works when we do not receive information about x_c would also work when do receive an observation if we choose to ignore the observation.

None of our decisions depend on $\xi(x'_c)$, thus no observation space mapping to S is necessary. □

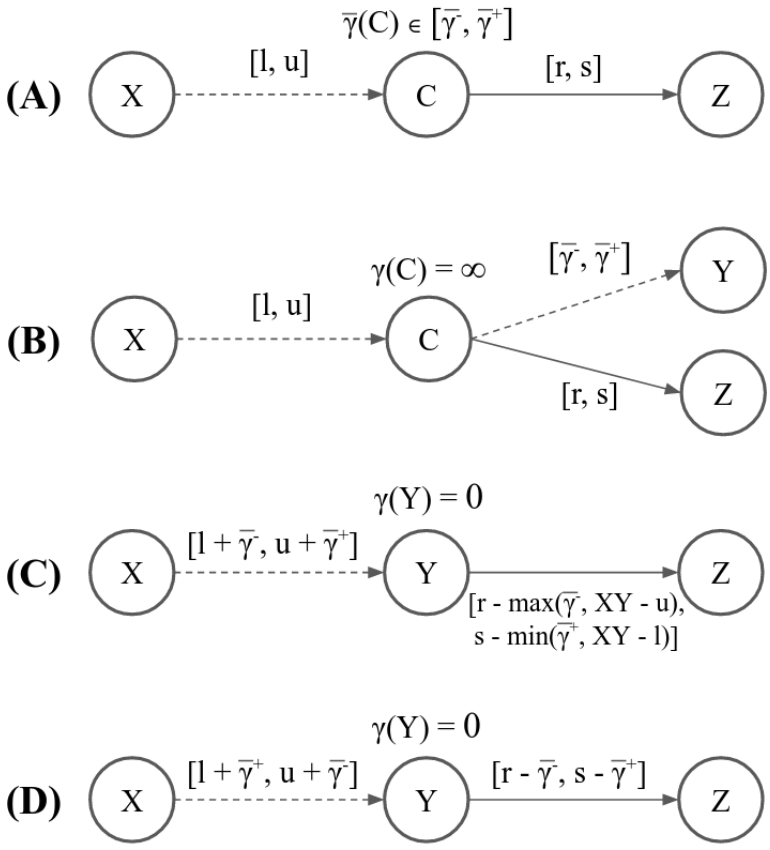


Figure 4-2: A visualization of the lemmas used to transform contingent links with variable observation delay and subsequent requirement links.

Lemma 9. *If $u - l \leq \bar{\gamma}^+(x_c) - \bar{\gamma}^-(x_c)$, we emulate $\bar{\gamma}(x_c)$ in S' using $\gamma(x'_c) = \infty$.*

Proof. We can ignore observations of x_c because they are not guaranteed to narrow where $\xi(x_c)$ was assigned in the range $[l, u]$.

Let α be the range of $\texttt{obs}(x_c)$ when $\xi(x_c) \in [l, l]$. Let β be the range of $\texttt{obs}(x_c)$ when $\xi(x_c) \in [u, u]$. By Equation 4.1,

$$\alpha = [l^-(x_c), l^+(x_c)]$$
$$\beta = [u^-(x_c), u^+(x_c)]$$

We can show that $u^-(x_c) \leq l^+(x_c)$.

$$u - l \leq \bar{\gamma}^+(x_c) - \bar{\gamma}^-(x_c)$$
$$u + \bar{\gamma}^-(x_c) \leq l + \bar{\gamma}^+(x_c)$$
$$u^-(x_c) \leq l^+(x_c)$$

The lower bound of β is less than the upper bound of α, thus $\alpha \cap \beta$. An observation $\texttt{obs}(x_c) \in [u^-(x_c), l^+(x_c)]$ could be the result of $\xi(x_c) = [l, l]$, $\xi(x_c) = [u, u]$, or any value $\xi(x_c) \in [l, u]$. Observations provide no information about the underlying contingent constraint, therefore we ignore $\texttt{obs}(x_c)$.

None of our decisions depend on $\xi(x_c')$, thus no observation space mapping to S is necessary. □

Lemma 10. *If $u - l > \bar{\gamma}^+(x_c) - \bar{\gamma}^-(x_c)$, we can emulate $\bar{\gamma}(x_c)$ under minimal information by replacing the bounds of x_c with $x_c' \in [l^+(x_c), u^-(x_c)]$ and letting $\gamma(x_c') = 0$.*

Proof. Under Lemma 10, observations $\texttt{obs}(x_c)$ are guaranteed to narrow the range of $\xi(x_c)$.

We have the same ranges for α and β as in Lemma 9, however we can show that $u^-(x_c) \geq l^+(x_c)$ instead.

$$u - l \geq \bar{\gamma}^+(x_c) - \bar{\gamma}^-(x_c)$$
$$u + \bar{\gamma}^-(x_c) \geq l + \bar{\gamma}^+(x_c)$$
$$u^-(x_c) \geq l^+(x_c)$$

Thus, receiving an observation is guaranteed to narrow the derived range of $\xi(x_c)$. The transformation tightens the range of x'_c to one where there is maximum ambiguity of the assignment of x_c while guaranteeing an execution strategy for any assignment of $x_c \in [l, u]$. □

After applying Lemma 10, despite the limited expected range of assignments in x'_c in S' compared to x_c in S, we can show that Lemma 13 guarantees a satisfying schedule for any $\texttt{obs}(x_c) \in [l^-(x_c), u^+(x_c)]$ using an $\mathcal{S}$ that employs *buffering* and *imagining* contingent events.

Definition 24. Buffering

Buffering a contingent event x_c is an execution strategy where, if x_c is observed earlier than the lower bound of the observation space $\texttt{obs}(x_c) < \omega_f^-(x'_c)$, we assign $\xi(x'_c)$ to the lower bound of the observation space, $\xi(x'_c) = \omega_f^-(x'_c)$.

Definition 25. Imagining

Imagining a contingent event x_c is an execution strategy where, if x_c is observed later than the upper bound of the observation space, $\texttt{obs}(x_c) > \omega_f^+(x'_c)$, we assign $\xi(x'_c)$ to the upper bound of the observation space, $\xi(x'_c) = \omega_f^+(x'_c)$.

Lemma 11. *If S' is fixed-delay controllable after applying Lemmas 10, 12, and 13 to contingent event Y with following requirement event Z, there is a valid $\mathcal{S}$ for any observation in the observation space of S, $\omega_v(Y) = [a^-(Y), b^+(Y)]$.*

Proof. We first note the observation space of S' is a subinterval of the original observation space of S, $\omega_f(Y') \subset \omega_v(Y)$, and there are two distinct ranges of observations that are not in $\omega_f(Y')$.

$$\omega_f(Y') = [a + \bar{\gamma}^+(Y), b + \bar{\gamma}^-(Y)];\ \omega_v(Y) = [a + \bar{\gamma}^-(Y), b + \bar{\gamma}^+(Y)]$$
$$\omega_f(Y') \not\supset [a + \bar{\gamma}^-(Y), a + \bar{\gamma}^+(Y)) \quad (\textit{"Early" observations})$$
$$\omega_f(Y') \not\supset (b + \bar{\gamma}^+(Y), b + \bar{\gamma}^+(Y)] \quad (\textit{"Late" observations})$$

We address the early observations first. The range of early assignments of $\xi(Y)$ in S that we care about are the ones that could produce an observation $\texttt{obs}(Y) \leq a + \bar{\gamma}^+(Y)$, which is $\xi(Y) = [a, a + (\bar{\gamma}^+(Y) - \bar{\gamma}^-(Y))]$. We rewrite the range of early assignments as $\xi(Y) = a + (\bar{\gamma}^+(Y) - \bar{\gamma}^-(Y)) - \epsilon$, where $0 \leq \epsilon \leq (\bar{\gamma}^+(Y) - \bar{\gamma}^-(Y))$. By the semantics of S, the range of assignments of $\xi(Z)$ is then,

$$\xi(Z) = [a + (\bar{\gamma}^+(Y) - \bar{\gamma}^-(Y)) - \epsilon, a + (\bar{\gamma}^+(Y) - \bar{\gamma}^-(Y)) - \epsilon] + [u, v]$$
$$\xi(Z) = [a + u + (\bar{\gamma}^+(Y) - \bar{\gamma}^-(Y)) - \epsilon, a + v + (\bar{\gamma}^+(Y) - \bar{\gamma}^-(Y)) - \epsilon]$$

The earliest assignment of Y' in S' is $\xi(Y') = a + \bar{\gamma}^+(Y)$. By the semantics of S', the range of assignments of $\xi(Z')$ is then,

$$\xi(Z') = [a + \bar{\gamma}^+(Y), a + \bar{\gamma}^+(Y)] + [u - \bar{\gamma}^-(Y), v - \bar{\gamma}^+(Y)]$$
$$\xi(Z') = [a + u + (\bar{\gamma}^+(Y) - \bar{\gamma}^-(Y)), a + v]$$

We see that $\xi(Z') \subseteq \xi(Z)$ for any ϵ, meaning the execution strategy when $\xi(Y') = a + \bar{\gamma}^+(Y)$ results in a valid assignment of $\xi(Z)$ for all early observations of $\xi(Y)$. We are safe to buffer early observations to $\xi(Y') = a + \bar{\gamma}^+(Y)$.

We use the same argument for imagining late observations. The range of late assignments of $\xi(Y)$ in S that we care about are the ones that could produce an observation $\texttt{obs}(Y) \geq b + \bar{\gamma}^-(Y)$, which is $\xi(Y) = b - (\bar{\gamma}^+(Y) - \bar{\gamma}^-(Y)) + \epsilon$. By the semantics of S, the range of assignments of $\xi(Z)$ is then,

$$\xi(Z) = [b - (\bar{\gamma}^+(Y) - \bar{\gamma}^-(Y)) + \epsilon, b - (\bar{\gamma}^+(Y) - \bar{\gamma}^-(Y)) + \epsilon] + [u, v]$$
$$\xi(Z) = [b + u - (\bar{\gamma}^+(Y) - \bar{\gamma}^-(Y)) + \epsilon, b + v - (\bar{\gamma}^+(Y) - \bar{\gamma}^-(Y)) + \epsilon]$$

The last assignment of Y' in S' is $\xi(Y') = b + \bar{\gamma}^-(Y)$. By the semantics of S', the range of assignments of $\xi(Z')$ is then,

$$\xi(Z') = [b + \bar{\gamma}^-(Y), b + \bar{\gamma}^+(Y)] + [u - \bar{\gamma}^-(Y), v - \bar{\gamma}^+(Y)]$$
$$\xi(Z') = [b + u, b + v - (\bar{\gamma}^+(Y) - \bar{\gamma}^-(Y))]$$

We see that $\xi(Z') \subseteq \xi(Z)$ for any ϵ, meaning the execution strategy when $\xi(Y') = b + \bar{\gamma}^-(Y)$ results in a valid assignment of $\xi(Z)$ for all late observations of $\xi(Y)$. In practice, there is no reason to wait until after $\texttt{obs}(Y) = b + \bar{\gamma}^-(Y)$ to receive a late observation. As soon as we see the clock has reached $b+\bar{\gamma}^-(Y)$, we are safe to imagine that $\texttt{obs}(Y)$ has been received. □

This concludes the modifications required to transform a contingent event $x_c \in X_c$ in S to its equivalent $x'_c \in X_c$ in S'. What remains is to address the transformation of requirement links, $x_r \in X_r$, in S such that their transformed equivalents, $x'_r \in X_r$ in S', express the same execution semantics in S' as they did in S. We will demonstrate the correctness of the transformations after Lemma 13.

Lemma 12. *If we have contingent link $X \Rightarrow C$ with duration $[l, u]$, outgoing requirement link $C \rightarrow Z$ with duration $[u, v]$ with an unobservable C, and contingent link $C \Rightarrow Y$ with range $[\bar{\gamma}^-(x_c), \bar{\gamma}^+(x_c)]$, we can emulate the role of the original requirement link during execution with a new link $Y \rightarrow Z$ with bounds $[u - max(\bar{\gamma}^-(x_c), XY - u), v - min(\bar{\gamma}^+(x_c), XY - l)]$, where XY is the true duration of $X \Rightarrow Y$.*

Proof. See Figure 4-2c for reference. 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 C 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 $X \Rightarrow C$ and $C \Rightarrow Y$ based on $X \Rightarrow Y$. We know that the lower-bound of $C \Rightarrow Y$ 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 link that limit its range to $[\bar{\gamma}^-, \bar{\gamma}^+]$. Taken together, during execution we can infer that the true bounds of $C \Rightarrow Y$ are $[max(\bar{\gamma}^-, XY - b), min(\bar{\gamma}^+, XY - a)]$. Since we have bounds only on Z's execution in relation to C, we can then infer a requirement link $Y \rightarrow Z$ 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 about X, we must be robust to any possible value assigned to $X \Rightarrow C$. This means that we would be forced to draw a requirement link $X \rightarrow Z$ 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 this strategy to the contingent link that we restricted in Lemma 10, in order to repair the remaining requirement links.

Lemma 13. *If we have an outgoing requirement link $C \rightarrow Z$ with duration $[u, v]$, where C is a contingent event, we can emulate the role of the original requirement link by replacing its bounds with $[u - \bar{\gamma}^-(x_c), v - \bar{\gamma}^+(x_c)]$.*

Proof. See Figure 4-2d for reference. If we directly apply the transformation from Lemma 12 and Figure 4-2c to our original STNU, we introduce complexity through the need to reason over *min* and *max* operations in our link bounds. However, from Lemma 10, we know that in a controllability evaluation context, it is acceptable for us to simplify the $X \Rightarrow Y$ link to a stricter range of $[a + \bar{\gamma}^+, b + \bar{\gamma}^-]$, instead of $[a + \bar{\gamma}^-, b + \bar{\gamma}^+]$. This means that for the purpose of evaluating controllability, we can assume $a + \bar{\gamma}^+ \leq XY \leq b + \bar{\gamma}^-$. When we evaluate the requirement link $Y \rightarrow Z$, we see $max(\bar{\gamma}^-, XY - b) = \bar{\gamma}^-$ and $min(\bar{\gamma}^+, XY - a) = \bar{\gamma}^+$. This gives us bounds of $[u - \bar{\gamma}^-, v - \bar{\gamma}^+]$ for the $Y \rightarrow Z$ requirement link as seen in Figure 4-2d. □

Lemma 13 handles outgoing requirement edges connected to contingent events. In addition, we must handle incoming edges.

Corollary 13.1. *If we have an incoming requirement link $Z \to C$ with duration $[u, v]$, where C is a contingent event, we can replace the bounds of the original requirement link with $[u + \bar{\gamma}^+(x_c), v + \bar{\gamma}^-(x_c)]$.*

Proof. A requirement link $Z \to C$ with bounds $[u, v]$ can be immediately rewritten as its reverse $C \to Z$ with bounds $[-v, -u]$. After reversing the edge, we can apply Lemma 13 to get $Y \to Z$ with bounds $[-v - \bar{\gamma}^-, -u - \bar{\gamma}^+]$, which we can reverse again to get $Z \to Y$ with bounds $[u + \bar{\gamma}^+, v + \bar{\gamma}^-]$. □

We can examine a concrete example of Lemmas 10, 12, and 13 to show equivalence in the transformation from Figure 4-2a to 4-2d. We start by building an example of 4-2a. Let $X \xRightarrow{[2,5]} C$ with $\bar{\gamma}(C) \in [1, 2]$ and $C \xrightarrow{[11,20]} Z$. If we learn of event C at time 4, then one possibility is that the realized duration of C could have been 2 with an observation delay of 2. In this case, event Z must be executed in $[13, 22]$. However, if the realized duration of C were 3 with an observation delay of 1, then Z would fall in $[14, 23]$. Given we cannot distinguish between the possibilities, we take the intersection of the intervals, yielding $Z \in [14, 22]$. Likewise, if we learn of C at time 6, then C could have been realized at time 5 with an observation delay of 1 or it could have been realized at time 4 with an observation delay of 2. In the first case, Z must then fall in $[16, 25]$, while in the second, Z would fall in $[15, 24]$. The intersection yields $[16, 24]$.

By the semantics represented in Figure 4-2d, we can build an equivalent network with $\gamma(Y) = 0$ by setting $X \xRightarrow{[4,6]} Y$ and $Y \xrightarrow{[10,18]} Z$. If Y is observed at time 4, Z must be executed in $[14, 22]$. If Y is observed at time 6, Z then must be executed in $[16, 24]$. The execution semantics for both cases match the equivalent networks from 4-2a described above.

4.4 Discussion

We have demonstrated a modeling formalism to describe temporal networks with uncertain observation delay, along with a sound and complete procedure for checking controllability of said temporal networks. VDC is sound because if it finds that S' has a valid execution strategy, then it must also be the case that S has an execution strategy. VDC is complete because if it finds that S' is not controllable, then there exists a projection of S that is uncontrollable, thus S is not variable-delay controllable.

4.5 Experimental Analysis

In this section, we provide empirical evaluations of our variable-delay controllability checking algorithms, showing that variable-delay controllability gives us a level of modeling expressiveness that cannot be captured by approximations that use delay controllability alone. We do so by constructing examples of variable-delay STNUs for realistic multi-agent coordination scenarios that are taken from the domain of planetary exploration, inspired by the real decision-making processes during Apollo EVAs and modern day EVA operations research. First, we briefly describe the operational environment, relevant actors, and decisions in EVAs. We then provide a selection of STNUs that reflect the activities and temporal constraints of planetary exploration. Using these building blocks, we make a case for the expressivity of VDC in modeling uncertain communication, then generate larger STNUs to demonstrate the soundness of variable-delay controllability checking.[1]

Now, we present a sample collection communication scenario in Figure 4-3 that is representative of the types of activities performed during exploration and requires uncertain communication delay to faithfully model.

At a high level, in this activity a crew of i astronauts perform j activities of scanning potential samples and receiving feedback from the science team as to whether they should store or discard those sample. Scanning requires liberating, that is chip-

[1]The implementation of the experiments herein can be found at [https://gitlab.com/mit-mers/delay-stnu-benchmarks].

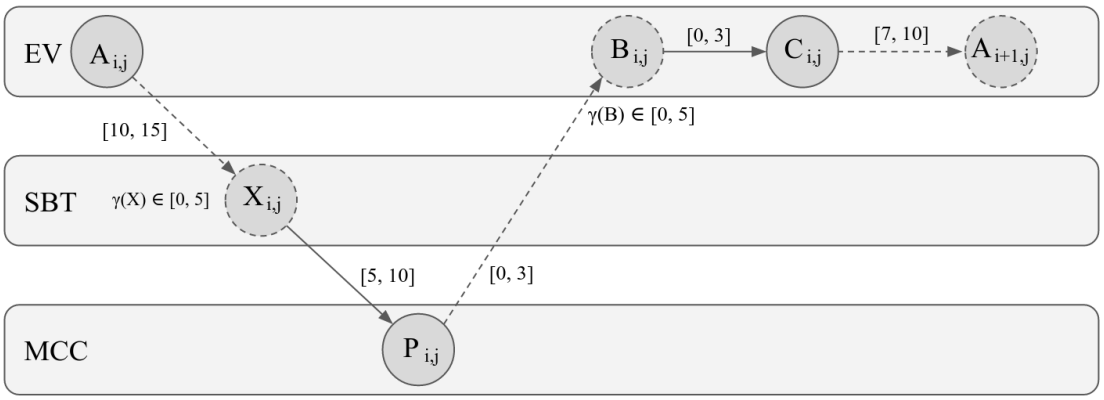


Figure 4-3: An STNU representing an EVA sampling task. The episode durations are representative of the bounds used in simulation. The depiction of this STNU with variable-delay is presented with rows representing actors to clarify the context of each event.

ping away, a piece of rock from an outcrop, $A_{i,j}$, and performing a scan of the newly exposed surface with a handheld spectrometer. Spectroscopy data is eventually received at $X_{i,j}$; we model this duration of this process with the contingent link $A_{i,j} \Rightarrow X_{i,j}$ where $X_{i,j}$ is uncontrollable because the time to liberate and scan is a function of the environment (eg. how hard the sample is to access), not the crew. The processing completion time of the handheld spectrometer is highly variable, and as such we have $\bar{\gamma}(X_{i,j})$ represent a variable delay in receiving the results of the scan. Interestingly, note that the general time of $A_{i,j}$ will be known immediately through the use of audio and video communications - the variability of $X_{i,j}$ refers to the delay of receiving the spectroscopy data itself.

During a narrow window of opportunity between the receipt of the sample information and a deadline imposed by Mission Control, $P_{i,j}$, the science team must confer and decide on a sample collection priority list to send to Mission Control, $X_{i,j} \rightarrow P_{i,j}$. Even once Mission Control has a sample priority list in hand, $P_{i,j}$, due to health and safety concerns, they may prioritize other messages before they send the science team's sampling priority decision to the crew. As such, the message passing process, $P_{i,j} \Rightarrow B_{i,j}$, is modeled as uncontrolled with a variable communication delay. Once the crew receives the priority list, B, they then stow the requested amount of samples

	Variable-delay controllable	Variable-delay uncontrollable
Min-fixed controllable	222	619
Min-fixed uncontrollable	0	159
Mean-fixed controllable	222	583
Mean-fixed uncontrollable	0	195
Max-fixed controllable	222	355
Max-fixed uncontrollable	0	423

Table 4.2: Variable-delay vs. minimum, mean, and maximum fixed-delay controllability results with the parallel installation STNU from Figure 5-3.

at $C_{i,j}$. Then the astronaut traverses to the next location and the procedure repeats anew. We use $C_{i,j} \Rightarrow A_{i,j+1}$ to model the time needed to traverse to the site of the next activity. We apply a requirement link with a lower-bound of 0 and an upper bound of the limiting consumable from the overall start of each STNU to its overall end after each astronaut has completed all activities.

With realistic STNUs in hand, we can now evaluate the performance of our variable-delay formulations. For the simulations presented in subsequent sections, we generated STNUs that follow the form of Figure 4-3 with randomized bounds on the links and delay functions, as will be described below.

We now will evaluate the comparative quality of variable-delay formulations against fixed-delay approximations by using the repeater installation scenario seen in Figure 5-3. We generate STNUs with four astronauts each performing five installations. We set the lower bounds of $A_{i,j} \Rightarrow B_{i,j}$ to 0 and choose the upper bounds from a uniform distribution of integers between 0 and 20, $\mathcal{U}_{[0,20]}$. There is no delay function for $B_{i,j}$. Likewise, for $B_{i,j} \rightarrow C_{i,j}$, we set the lower bounds to 0 and choose an integer upper bound in $\mathcal{U}_{[0,15]}$. $C_{i,j} \Rightarrow D_{i,j}$ has a lower bound of 0 and an upper bound integer chosen in $\mathcal{U}_{[0,20]}$. The variable-delay function $\gamma(D_{i,j})$ has a lower bound of 0 and upper-bound chosen from the exponential distribution $f(t) = \lambda e^{-\lambda t}$ with $\lambda = 3$. $D_{i,j} \Rightarrow A_{i,j+1}$ takes a lower bound integer, a, from $\mathcal{U}_{[10,20]}$ and its upper bound in $a + \mathcal{U}_{[4,10]}$. Lastly, we pick a random limiting consumable as the multiple of the number of activities and an

integer from $\mathcal{U}_{[50,60]}$.

We employ three different strategies for each $\gamma(x_c)$ in S 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 4.2). Note that our randomly generated variables, notably the choice of $\gamma(C_{i,j})$ and the width of the following $C_{i,j} \rightarrow D_{i,j}$ link, were selected such that the STNUs generated could be variable-delay, fixed-delay, dynamic, or strong controllable, or uncontrollable. 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 80.0%. The mean and maximum fixed-delay approximations have more reasonable false positive rates at 74.9% and 45.6% respectively. Since all approximations yield the correct answer when the original STNU is variable-delay controllable, it follows 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 dependent on the width of the variable-delay ranges found in the network. We can increase the likelihood that a delay takes longer by increasing the choice of λ in our exponential delay function. When we vary our delay function using $\lambda = 4.5, 6, 7.5$, and 9, the false positives of the max-delay approximation are 27.9% 12.9%, 7.0%, and 3.1%, respectively.

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 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

	Variable-delay controllable	Variable-delay uncontrollable
Elongated controllable	36	0
Elongated uncontrollable	186	778

Table 4.3: Variable-delay controllability vs. the controllability of a network that elongates its contingent links to account for observational uncertainty when using an exponential delay function with $\lambda = 3$.

about the difference in uncertainties between the original event and the observation of that event.

As seen in Table 4.3, this approach yields no false positives, but still presents a modestly high false negative rate of 19.3%. 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.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Scheduling Events Despite Uncertain Observations

Now that we have shown there exists a valid execution strategy for variable-delay controllable STNUs, we contribute a novel scheduling and dispatching architecture for online, dynamic execution. In this Chapter, our aim is to describe the single-agent form of a new instantiation of Kirk, *Delay Kirk*, that can reason over uncertain observation delay to decide when to execute requirement events on real hardware. There are two main components to Delay Kirk: (1) a *delay scheduler*, and (2) a *delay dispatcher*. As will be shown in Section 5.2, scheduling variable-delay controllable STNUs is an extension to existing dynamic scheduling algorithms with modifications for the execution strategy shown to be sound and complete in Chapter 4.

Additionally, to the best of our knowledge, scheduling fixed-delay STNUs has not been presented in the literature. Fixed-delay scheduling is required for addressing (1). As such, we contribute a fixed-delay scheduler in Section 5.2. The execution strategy from Chapter 4 will be shown to be a small extension to the fixed-delay scheduler. As to (2), to the best of our knowledge, there are no other formalized dispatching algorithms in the literature. In the development of Delay Kirk, we found it to be extremely useful to formalize dispatching as part of creating a clear interface boundary between scheduling and dispatching. The dispatching algorithms we put forth in Section 6.1.2 represent novel contributions to temporal reasoning.

This Chapter makes additional contributions to scheduling and dispatching. Safely executing events on real hardware requires modifications to generating decisions in dynamic scheduling. We include said modifications, with confluent interfaces in dynamic dispatching, to suit our intended use cases for Delay Kirk.

Finally, Section 5.3 provides a series of benchmarks of the scheduling and dispatching algorithms described in this Chapter.

5.1 Dynamic Scheduling through Real-Time Execution Decisions

We first provide a necessary overview of dynamic scheduling of vanilla STNUs, which we will extend for STNUs with observation delay in Section 5.2.

An STNU, S, that exhibits dynamic controllability can be *scheduled* dynamically (or *online*). At a high-level, dynamic scheduling is the process of mapping the history of event assignments to the execution time of future free events. We will build off of the scheduling work by Hunsberger [14], [15], which describes an $O(N^3)$ online procedure, FAST-EX, for dynamic scheduling of STNUs. We chose FAST-EX because, to the best of our knowledge, this is the fastest dynamic scheduling algorithm in the literature today. At its core is the notion of *Real-Time Execution Decisions* (RTEDs), which map a time to a set of requirement events to be executed and are generated based on *partial schedules* of STNUs being executed. `WAIT` decisions may also be produced, reflecting the need to wait for the assignment of a contingent event before continuing. RTED-based scheduling applies a dynamic programming paradigm in three steps:

1. creating a dispatchable form of temporal constraints offline in the form of a distance graph,
2. updating the dispatchable form as the partial schedule is updated online through event assignments, and
3. querying the dispatchable form online to quickly find the next RTED [34].

The dispatchable form employed by FAST-EX is the *AllMax* distance graph, which is first described in the Morris $O(N^4)$ DC-checking procedure [31].

Definition 26. AllMax Distance Graph [32]

The *AllMax* distance graph is a distance graph exclusively consisting of unlabeled and upper-case edges.

The key idea of FAST-EX is maintaining accurate distances from an artificial zero point, Z, of the distance graph to all events. At the outset of execution, all events from S are present as nodes in *AllMax*. As events are assigned, *AllMax* performs update steps using Dijkstra Single Source/Sink Shortest Path (SSSP) to maintain distances to unexecuted events, while also collapsing executed events to Z. We include pseudo-code of the real-time update step in Figure 2.

Input: Time t; Set of newly executed events $\texttt{Exec} \subseteq X_e \cup X_r$; AllMax Graph G; Distance matrix D, where $D(A, B)$ is the distance from A to B
Output: Updated D

FAST-EX Update:

1 **for** *each contingent event* $C \in$ ***Exec*** **do**
2 Remove each upper-case edge, $Y \xrightarrow{C:-w} A$, labled by C;
3 Replace each edge from Y to Z with the strongest replacement edge;
4 **end**
5 **for** *each event* $E \in$ ***Exec*** **do**
6 Add lower-bound edge $E \xrightarrow{-t} Z$;
7 **end**
8 For each event X, update $D(X, Z)$ using Dijkstra Single-Sink Shortest Paths;
9 **for** *each event* $E \in$ ***Exec*** **do**
10 Add upper-bound edge $Z \xrightarrow{t} E$;
11 **end**
12 For each event X, update $D(Z, X)$ using Dijkstra Single-Source Shortest Paths;

Algorithm 2: Algorithm for updating distances for all events in relation to Z upon the execution of an event. Adapated from [3], Fig. 19.

With an up-to-date distance graph in hand, we can perform an online query for the current RTED.

Definition 27. Real-Time Execution Decisions [34]

A *Real-Time Execution Decision* is a two-tuple $\langle t, \chi \rangle$, where:

- t is a time with domain $\mathbb{R}$,
- χ is a set of $x_r \in X_r$ to be executed at time t

Let U_x be the set of unexecuted free timepoints. If U_x is empty, then the RTED is to `WAIT`. Otherwise, we find the lower bound of the earliest executable time point and the set of executable events associated with it.

$$t = \min\{-D(X, Z) \mid X \in U_x\} \tag{5.1}$$

$$\chi = \{X \in U_x \mid -D(X, Z) = t\} \tag{5.2}$$

We cannot execute events in the past. Let `now` be the current time, i.e. the last timepoint captured in the event assignments. It is possible that $t \leq$ `now`, in which case we must reassign t to guarantee that $t >$ `now`. To do so, we update t as follows, where t^+ is earliest upper bound of the executable timepoints,

$$t^+ = \min\{D(Z, X) \mid X \in U_x\} \tag{5.3}$$

$$t = \frac{\texttt{now} + t^+}{2} \tag{5.4}$$

So long as $t^+ >$ `now`, we know that the reassignment of t ensures $t >$ `now`.

5.2 Delay Scheduling as an Extension to Dynamic Scheduling

Figure 5-1 presents a high-level overview of the information flow in the scheduling process.

In order to schedule a variable-delay STNU, the core problem we must address is that, to date, there is no means to directly create a corresponding dispatchable form

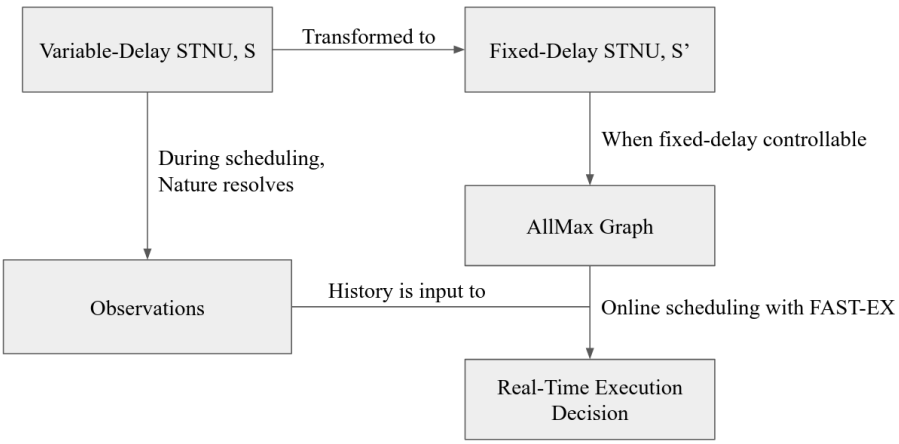


Figure 5-1: A high-level flow chart showing how we use variable-delay STNUs to generate scheduling decisions. The boxes represent the data structures involved in scheduling, while the arrows are the processes that are followed to eventually produce RTEDs.

that accounts for uncertain assignments resulting from variable observation delay. We encountered this same problem when describing the process of checking VDC in Section 4.3. We overcame this limitation by first transforming the variable-delay STNU to a fixed-delay STNU before checking FDC. A similar strategy will be followed for scheduling in that we will transform the variable-delay to a fixed-delay STNU, then dispatch events using the dispatchable form of the fixed-delay STNU instead. However, doing so creates a second problem. While we will be performing FAST-EX against the fixed-delay STNU, the contingent event observations we receive will adhere to the constraints and variable-delay function of the variable-delay STNU. Hence, we must modify our real-time update and RTED generation algorithms to account for early and late contingent event observations.

We start by providing an explanation of fixed-delay scheduling, before expanding it to address the execution strategies of variable-delay scheduling.

5.2.1 Fixed-Delay Scheduling

We first establish the algebra of receiving observations.

Lemma 14. *For any contingent event, $x_c \in S$ or $x'_c \in S'$, observing x_c at time*

$t \in [l^-(x_c), u^+(x_c)]$ *fixes the observation to* $\texttt{obs}(x_c) = [t, t]$.

Proof. Prior to execution, an observation of x_c may fall anywhere within the set-bounded interval from the earliest possible observation at $l^-(x_c)$ to the last possible observation at $u^+(x_c)$. Receiving an observation $\texttt{obs}(x_c) = t$ during execution eliminates all possible observations outside the interval $[t, t]$. □

Lemma 15. *For any temporal constraint, x, with bounds $x \in [l, u]$ for some l and u, and timepoint $t \in [l, u]$, if information reduces the bounds of x to $x \in [t, t]$, we may assert $x = t$.*

Proof. When the bounds of an interval, $x \in [l, u]$ are fixed such that $t = l = u$, we can assert that x must have resolved to t. □

Lemma 16. *For any contingent event $x'_c \in X_c$ in fixed-delay controllable S', if $\gamma(x'_c) \in \mathbb{R}$, we assign $\xi(x'_c) = \texttt{obs}(x_c) - \gamma(x'_c)$ in the dispatchable form of S'.*

Proof. The central challenge of checking fixed-delay controllability is determining that an execution strategy exists that allows an agent to wait an additional $\gamma(x'_c)$ time units after a contingent event has been assigned to learn its outcome. Importantly, the γ function is not used to modify the edges of the labeled distance graph, which are derived from the constraints $r \in R_e \cup R_c$ in S'.

As $\gamma(x'_c)$ resolves to a known and finite value, we can derive the true value of $\xi(x'_c)$ to be assigned in the labeled distance graph. Contingent event assignments are recorded in the labeled distance graph as follows, where $\texttt{obs}(x_c)$ is the resolved observation,

$$\xi(x'_c) = \texttt{obs}(x_c) - \gamma(x'_c) \tag{5.5}$$

□

The FAST-EX real-time update algorithm, Algorithm 2, then becomes Algorithm 3.

No other modifications to FAST-EX are required to schedule a fixed-delay STNU.

Input: Time t; Set of newly observed events $\texttt{Exec} \subseteq X_e \cup X_r$; AllMax Graph G; Distance matrix D, where $D(A, B)$ is the distance from A to B; Fixed-delay function γ;

Output: Updated D

FAST-EX Update with Fixed Observation Delay:

for *each contingent event* $C \in \texttt{Exec}$ **do**
  $\xi(C) \leftarrow \texttt{obs}(C) - \gamma(C)$;
  Remove each upper-case edge, $Y \xrightarrow{C:-w} A$, labled by C;
  Replace each edge from Y to Z with the strongest replacement edge;
end
for *each event* $E \in \texttt{Exec}$ **do**
  Add lower-bound edge $E \xrightarrow{-t} Z$;
end
For each event X, update $D(X, Z)$ using Dijkstra Single-Sink Shortest Paths;
for *each event* $E \in \texttt{Exec}$ **do**
  Add upper-bound edge $Z \xrightarrow{t} E$;
end
For each event X, update $D(Z, X)$ using Dijkstra Single-Source Shortest Paths;

Algorithm 3: Algorithm for updating distances for all events in relation to Z upon the execution or observation of an event.

5.2.2 Variable-Delay Scheduling

Our execution strategy must address each of the following special categories of contingent event observations:

1. contingent events with infinite observation delay,
2. contingent events that are observed outside $[l^+(x_c), u^-(x_c)]$ in S'.

The first category is a requirement for dispatching the fixed-delay equivalent of a variable-delay STNU. If the constraints of a problem domain are modeled directly in a fixed-delay STNU and the modeler gives a contingent event, x_c, infinite delay, e.g. $\gamma(x_c) = \infty$, the event will never be observed and thus a fixed-delay scheduler has no need for an execution strategy in the event that x_c is observed. However, by Lemmas 8 and 9 there are some contingent events with potentially finite observation delay in S that are transformed to infinite observation delay in S', making it possible that the scheduler receives observations of them.

Lemma 17. *For any contingent event $x'_c \in X_c$ in fixed-delay controllable S', if $\gamma(x'_c) = \infty$, we mark the event executed but do not assign $\xi(x'_c)$ in the dispatchable form of S'.*

Proof. If we are scheduling a fixed-delay STNU, S', that is already known to be fixed-delay controllable, an execution strategy must exist that is independent of the assignment of $\xi(x'_c)$ when $\gamma(x'_c) = 0$. We are not required to record $\xi(x'_c)$ when $\gamma(x'_c) = \infty$ to guarantee controllability and may safely ignore it.

We mark the event executed to prevent it from appearing in future RTEDs. □

The second category refers to the need for buffering and imagining events as a result of Lemma 10 using the execution strategy proven to be valid in Lemma 11. There are three regimes of contingent event observations to address.

1. $\texttt{obs}(x_c) \in [l^-(x_c), l^+(x_c))$, ie. strictly earlier than the range of $\xi(x'_c)$,
2. $\texttt{obs}(x_c) \in [l^+(x_c), u^-(x_c)]$, ie. the range equivalent to x'_c, and
3. $\texttt{obs}(x_c) \in (u^-(x_c), u^+(x_c)]$, ie. strictly later than the range of $\xi(x'_c)$.

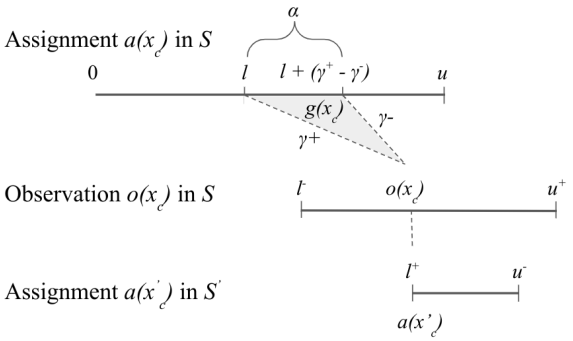


Figure 5-2: Here, we show how the combination of $\xi(x_c)$ and $\bar{\gamma}(x_c)$ lead to an assignment of $\xi(x'_c)$ in S'. We see the range $\alpha \in [l, l + \bar{\gamma}^+(x_c) - \bar{\gamma}^-(x_c)$ representing the earliest and latest assignments of $\xi(x_c)$ that could result in $\texttt{obs}(x_c) \in \xi(x'_c) \in [l^+(x_c)$, l^+(x_c)]$. The grey region represents the range of possible observation delays, $\bar{\gamma}(x_c)$, supporting $\xi(x'_c) \in [l^+(x_c), l^+(x_c)]$.

Note that we omit the $-\gamma(x'_c)$ term from Equation 5.5 in this analysis due to the fact that $\gamma(x'_c) = 0$ after applying Lemma 10.

Our execution strategy is to then make the following assignments during the FAST-EX real-time update.

$$\xi(x'_c) = \begin{cases} l^+(x_c) & \text{if } \texttt{obs}(x_c) \in [l^-(x_c), l^+(x_c)) \text{ } (\textit{buffering}) \\ \texttt{obs}(x_c) & \text{if } \texttt{obs}(x_c) \in [l^+(x_c), u^-(x_c)] \\ u^-(x_c) & \text{if } \texttt{obs}(x_c) \in (u^-(x_c), u^+(x_c)] \text{ } (\textit{imagining}) \end{cases} \tag{5.6}$$

In the first case, we cannot immediately schedule buffered events. It may be the case that there are other unexecuted timepoints between $\texttt{obs}(x_c)$ and $l^+(x_c)$. If we make an assignment at $l^+(x_c)$, we would be preempting later timepoints, which would cause us to later make assignments in the past, which invalidates our assumptions of partial history. Thus, we buffer x'_c in the sense that we must wait until $l^+(x_c)$ to assign $\xi(x'_c) = l^+(x_c)$.

In the last case, late observations are assigned to an earlier time. During execution, time is always increasing. There is no need to wait to make an observation after $u^-(x_c)$. Instead, we modify RTED generation, namely Equation 5.1, such that we dispatch x'_c at $u^-(x_c)$ if it is not been observed before $u^-(x_c)$. Let U_c be the set of unobserved contingent timepoints.

$$t_x = \min\{-D(X, Z) \mid X \in U_x\} \tag{5.7}$$

$$t_c = \min\{D(Z, X) \mid X \in U_c\} \tag{5.8}$$

$$t = \min\{t_x, t_c\} \tag{5.9}$$

$$\chi_x = \{X \in U_x \mid -D(X, Z) = t\} \tag{5.10}$$

$$\chi_c = \{X \in U_c \mid D(Z, X) = t\} \tag{5.11}$$

$$\chi = \chi_x \cup \chi_c \tag{5.12}$$

We see that RTEDs may now include unobserved (or unexecuted) contingent timepoints at their upper bounds. Note that there is no need to distinguish between contingent events that are the result of tightening during the fixed-delay transformation by applying Lemma 10 and others. We assume that the contingent constraints of the variable-delay STNU accurately reflect Nature. The latest any other contingent event should be observed is their upper bound in S' and thus should never be in the set of events, χ, of an executed RTED.

We have defined variable-delay execution strategies for when contingent events have infinite delay and tightened constraints. The remaining category of contingent events is when a contingent event has a finite, non-zero $\gamma(x'_c)$ in S'. If that is the case, x'_c must have had fixed observation delay in S, Lemma 7, and can be scheduled normally after backing out the observation delay with Equation 5.5.

We have addressed the key issue of reconciling observations from S with the dispatchable form from S'. Section 6.1 will present a dispatcher and wrapper algorithms on top of FAST-EX that combine to add robustness for variable observation delay.

5.3 Experimental Analysis

We first introduce an example which models a construction task on the lunar surface that will be used to randomly generate STNUs with realistic constraints for benchmarking purposes. We then describe benchmarks against the performance of

the real-time FAST-EX update with the variable-delay execution strategy, the dispatching routine, and observations. All benchmark code can be found at `https://gitlab.com/enterprise/enterprise` in the `kirk-v2/benchmarks` directory.

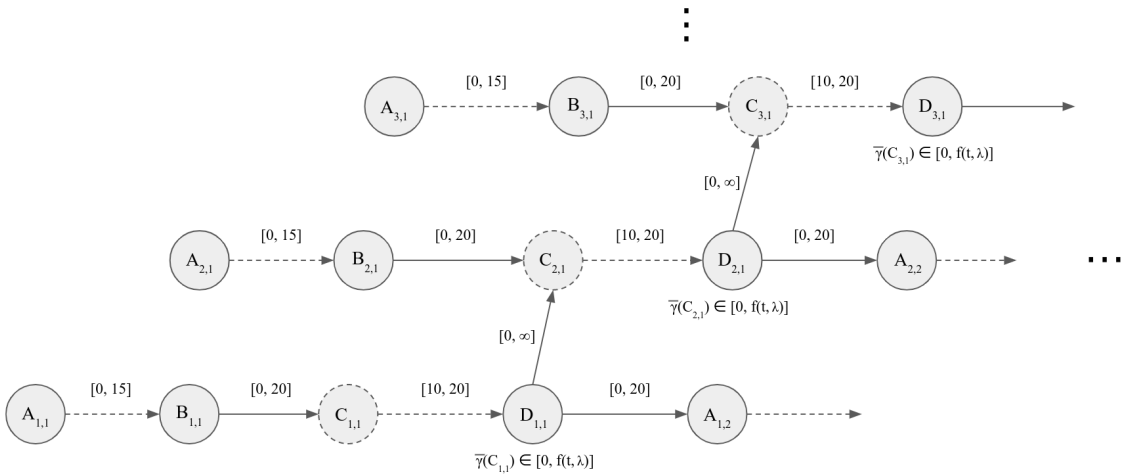


Figure 5-3: An STNU representing the installation and test of repeater antennas. Each row represents a single rover. The episode durations are representative of the bounds used in simulation.

It is possible that, before NASA is ready to grow the population of a lunar base, there is a need to prepare a communications infrastructure near a habitat with a large grid of repeater antennas. This scenario depicted with the STNU in Figure 5-3 represents an installation task wherein i rovers (mobile robot) are each installing j surface signal repeater antennas. During the activity, every rover is responsible for installing one repeater. Each event, X, is represented for the i-th rover and j-th repeater as $X_{i,j}$. All numbers in the figure are representative of the minimum and maximum of the randomly generated constraints in the benchmarks.

The rovers work in parallel, with a $[0, \infty)$ requirement link from the start of the STNU to each $A_{i,1}$ (not shown). The first episode, $A_{i,j} \Rightarrow B_{i,j}$, represents traversing to the site of the installation. We model traverses as uncontrollable due to the fact that crews are embarking across unknown terrain. Once at the site, an antenna is installed as represented by $B_{i,j} \rightarrow C_{i,j}$. Each repeater needs to have its configuration tested and confirmed working by $D_{i,j}$, represented by the edge $C_{i,j} \Rightarrow D_{i,j}$. Confir-

mation takes the form of a request-response cycle to the ground. We model $D_{i,j}$ as uncontrolled and with variable delay because each antenna takes an unknown time to self-configure and the crew does not know when they will receive a response from Earth that the repeater installation has been verified due to uncertainty in communication. Bandwidth is limited, so we limit the number of repeaters simultaneously sending requests to their configuration. We use the $D_{i,j} \to C_{i+1,j}$ links to enforce that the start of the confirmation of the next repeater does not begin until after the previous repeater's confirmation. Confirmations are required until we reach the last crew member or the last activity. Once testing is complete, the rovers clean up their workstations, $D_{i,j} \to A_{i,j+1}$ and then repeat the cycle until all antennas have been installed.

To perform the benchmarks, we generated variable-delay STNUs of increasing sizes with randomly determined constraints as previously described. We immediately checked VDC of each STNU, and would generate new STNUs of a given size until we found one that was confirmed to be VDC. We then simulated scheduling and dispatching of the STNU with a faster-than-realtime clock. No driver was present, so all real events were scheduled immediately.

These data were collected on an Intel i7-10710U 6c/12t mobile processor with 16GB of RAM in a ThinkPad X1 Carbon Gen 7 laptop. All tests were run while the laptop was attached to wall power. The code was written in Common Lisp and all benchmarks were run with Steel Bank Common Lisp version 2.0.1. To reduce the time spent running benchmarks, we scheduled multiple STNUs in parallel, with each STNU being scheduled in its own thread.

The regressions below were performed using the Python packages `scipy` [35] and `sklearn` [36], then graphed with `matplotlib` [37].

The implementation of the delay scheduler from which these data were collected has a bug that we have been unable to identify. We have only seen the bug surface with STNUs with more than about 50 events. The bug takes effect when observing a contingent event, x_c, which has incoming contingent constraint $[l, u]$. If we observe x_c at some time t, where $l \leq t < u$, the Dijkstra SSSP subroutine may unexpectedly find

a negative edge and raise an error. We have been able to replicate the problem for specific STNUs with specific observations, and, as of the time of this writing, we are still investigating the cause. We do not believe it meaningfully impacts the validity of the benchmarks below.

5.3.1 Scheduling

We start with the runtime performance of schedule updates. There can be runtime variance for each individual call to the scheduling update routine, so we focus on the total time spent scheduling all events in the STNU. According to the FAST-EX algorithm, the total runtime is dominated by the $O(N \log N)$ runtime of Dijkstra SSSP, where N is the total number of events. Thus, the total runtime to schedule every event in an STNU is $O(N^2 \log N)$ [15, p. 144]. Given the changes we made to FAST-EX are also dominated by Dijkstra SSSP, we expect to see the same runtime performance here.

Figure 5-4 clearly shows that the total time spent scheduling STNUs with $N \leq 300$ follows $O(N^2 \log N)$ as expected, with a coefficient of determination for the regression of $R^2 = 0.995$.

If we expand the size of STNUs to $N \leq 600$, then we see the total runtime correspond less closely with $O(N^2 \log N)$, as can be seen in Figure 5-5. We believe the deviation is due to programming language features in lisp outside of our control, such as automated memory management.

5.3.2 Event Observations

Next, we examine the runtime characteristics of event observations. While generating VDC STNUs, we also collected possible ranges of time to observe the confirmation event. As scheduling progressed, we automatically triggered observations of the confirmation event at a time randomly selected within the range given.

Contingent event observations are made much less frequently than scheduling. While we must schedule every event in an STNU, our benchmarking procedure will

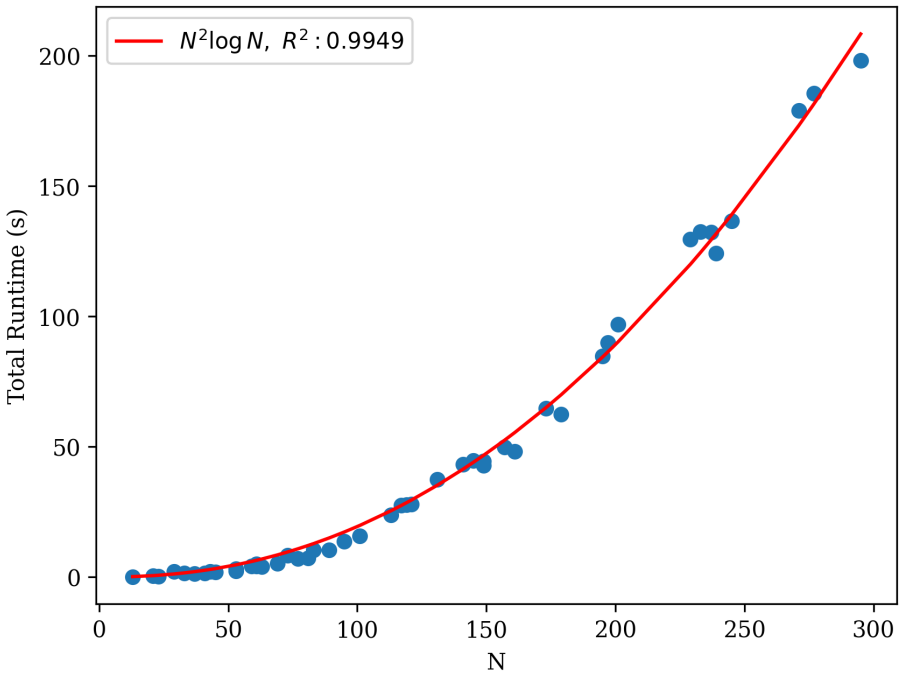


Figure 5-4: Total runtime data for scheduling all events in VDC STNUs where $N \leq 300$.

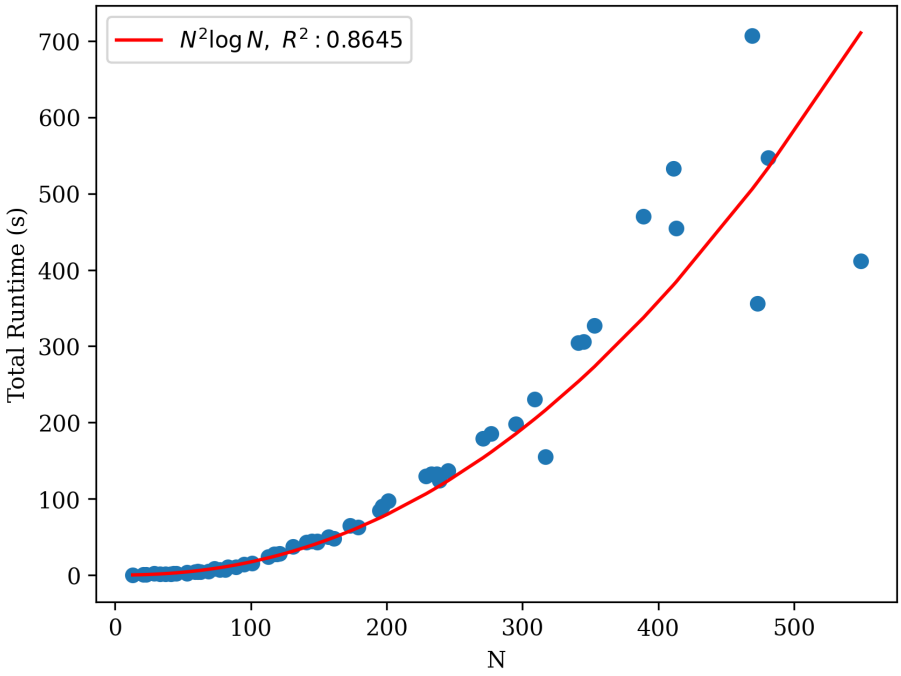


Figure 5-5: Total runtime data for scheduling all events in VDC STNUs where $N \leq 600$.

only observe a small fraction of the events. As a result, sample sizes are small. Given event observations are dominated by the call to FAST-EX for a scheduling update, we expect to see runtimes on the order of $O(N \log N)$. However, the data in Figure 5-6 show significant deviation from it. Given that the method call to observe events is a thin wrapper around a FAST-EX update, we believe the error of this graph is due to small sample sizes.

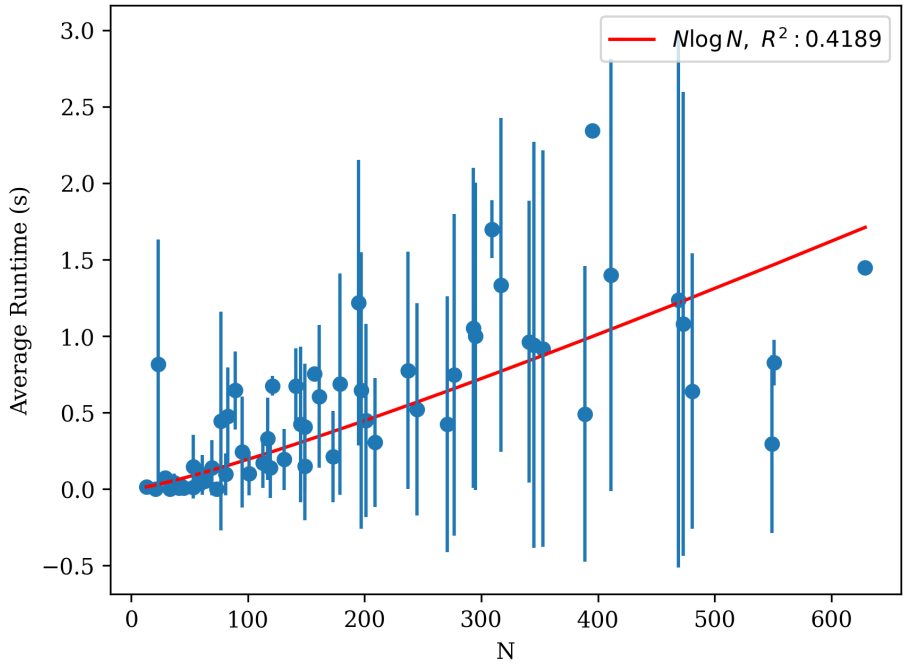


Figure 5-6: Average runtime data for observing events in VDC STNUs. Error bars represent standard deviation.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

An Executive for Scheduling with Observation Delay

A delay scheduler is only useful if there is a system capable of dispatching the actions associated with RTEDs. A delay scheduler could be integrated as a subsystem for a digital assistant, if such a digital assistant has a means for surfacing RTEDs to a user in a useful manner. Instead, for this thesis, we choose to integrate with a high-level task and motion planner, *Kirk* [38]. Kirk is a complete, end-to-end executive in that it can take human-friendly problem specifications as input and send commands to hardware as output.

At a high-level, Kirk operates by first taking a description of the problem domain as written by domain experts, which should include the constraints, agent dynamics, environment, and starting and goal states of the problem at hand. Kirk then generates state plans, which consist of episodes that organize the occurrence of events as activities. State plans may include temporal constraints and non-temporal constraints, such as classical planning preconditions and effects. Kirk checks plans for consistency using an optimal satisfiability (OpSAT) solver [39]. Next, it elaborates temporal plan networks (TPNs) [40] to sub-executives when it encounters constraints and goals it cannot plan against directly. Finally, it dispatches actions to hardware. For the purpose of this thesis, we focus on Kirk's capability to dispatch actions from state plans.

Below, we present an instantiation of Kirk, *Delay Kirk*, designed to dispatch actions from state plans with uncertain observations. A delay scheduler lives at the core of Delay Kirk, with a new component, a *delay dispatcher* taking responsibility for translating RTEDs into actions in the real world. In this chapter, we start by defining a delay dispatcher, which plays a key role in enabling delay scheduling. Next, we present a high-level overview of Kirk's architecture. Finally, we define necessary components of an input language, the Reactive Model-Based Programming Language (RMPL), used to represent constraints and action models for Delay Kirk. We present experimental results on the performance of the delay dispatcher, however, experimentation with a full Delay Kirk executive are presented in Section 7.3.

6.1 Dynamic Dispatching of STNUs with Observation Uncertainty

We assume that events in an STNU map one-to-one with actions in the real world. To put the design of the dispatcher in context, it is worth considering what events may look like. In the case of a robotic agent, requirement events may represent the instantaneous timepoints when motion plans begin, while contingent events could be anything from the completion of said motion plans to the receipt of `PROCEED` messages from a third party. For a human, requirement events could be presented in a mission timeline as the start of planned actions such as the collection of scientific samples. The end of a sampling activity would then be an uncontrollable event. Or uncontrollable events could be the actions performed by other agents, like say another astronaut on an EVA, with whom temporal constraints are shared. In both the case of the robot and the human, a robust dispatcher should take into consideration that passing a message to the agent telling it to execute a requirement event does not cause the event to occur instantaneously. Put in other words, we are not guaranteed that dispatching an action causes an event to be scheduled instantaneously. A robot may require offline processing before it executes the motion plan. Or a human may need

to acknowledge that they have started the action their digital assistant has instructed them to perform. Neither situation is a problem for our chosen formalism for temporal reasoning so long as each requirement event is assigned at some point within their constraints in the STNU. In our view, the dispatcher is responsible for ensuring requirement constraints are met by both monitoring the real-world and interfacing with hardware to cause actions to be performed.

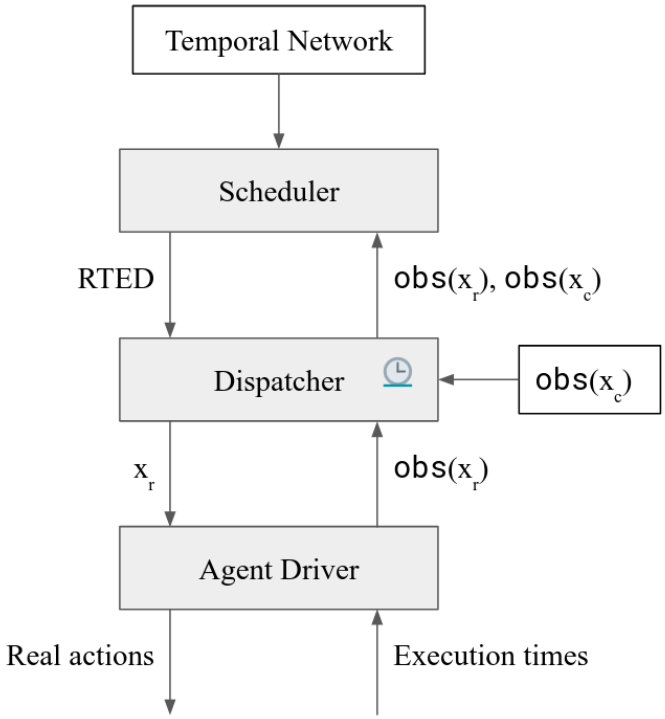


Figure 6-1: A more detailed view of the delay dispatcher architecture.

As depicted in Figure 6-1, there are two key pathways within a delay dispatcher. First, a delay dispatcher takes an RTED as input and causes actions to happen in the environment as output. Second, it takes event observations as input and may cause the scheduler to record events as output.

Additionally, we introduce a sub-component, a *driver*, that can interpret dispatched events and cause some action to be performed in the environment. We separate the driver and delay dispatcher in Figure 6-1 for completeness, however, for all intents and purposes, the driver is a sub-component of a delay dispatcher. A driver is defined as performing the following transformation: given some event, x, as input

the driver should cause an action to be performed. For instance, if Delay Kirk is controlling a robotic manipulator over ROS, the driver would receive an event string as input and publish ROS messages as output.

In this Section, we contribute a set of algorithms for building the dispatcher for a robust executive that can reason over observation delay and safely enact the actions symbolized in requirement events in the real world. We focus on the interpretation, management, and flow of RTEDs in Section 6.1.2. In Section 6.1.3 we describe the process of observing events. But first, we present a novel view on RTEDs that is required for dispatching events to real hardware in Section 6.1.1.

6.1.1 Guaranteeing Agents Receive Actionable Events

We take the view that events in an STNU may be interpreted as commands by the driver. It is improper to knowingly send an invalid command. Accordingly, the driver must never receive an event (in an RTED) that cannot be mapped to a corresponding action in its environment. As such, it is the dispatcher's responsibility to filter events in order to only dispatch valid actions.

In a variable-delay STNU, there are events that are associated with actions and there are events that are not. We call these *real* and *no-op* ("no operation") events. Only controllable events may be real, but both uncontrollable *and* controllable events may be no-op. Below, we present our rationale for the distinction between real and no-op events, and how we modify real-time execution decisions accordingly.

To start, imagined uncontrollable events are no-ops. They are assignments we artificially perform with no corresponding real-world action, and solely exist to maintain the controllability of the fixed-delay dispatchable form. Imagined events should never be dispatched to a driver.

There are requirement events that are also no-ops. Consider the process of normalization of an STNU [31]. While building the labeled distance graph during a DC check, we rewrite contingent links such that their lower bounds are always 0. For instance, for an uncontrollable event C and requirement event E, $C - E \in [l, u]$, during normalization we create a new requirement event, C', fixed at the lower bound of the

contingent link, and then shift the bounds of the contingent link to start at 0 while maintaining the original range, $u - l$. This results in two constraints: $E - C' \in [l, l]$ and $C - C' \in [0, u - l]$. The original contingent link's semantics are thus maintained.

Importantly, the requirement events representing the normalized lower bounds of uncontrollable events are in the dispatchable form for dynamic scheduling because we draw the AllMax graph directly from the DC check. To a scheduler, there is no distinction between the semantics of a real event, as modeled by a human planner writing an STNU for an agent to execute, and C', an artifact of checking controllability. Both are modeled in the AllMax distance graph forming the basis of RTED generation. However, an agent cannot dispatch any action to satisfy $E - C'$, rather C' should simply be scheduled at the appropriate time. Thus, we make the following addendum to the definition of RTEDs.

Definition 28. Event-No-op Pair

An *Event-No-op Pair*, *event-noop*, is a two-tuple, $\langle x, noop \rangle$, where:

- x is an event in $X_e \cup X_c$,
- *noop* is a boolean, where if true, the event cannot be interpreted by the driver, else the event is a valid command.

Definition 29. RTED with Operational Distinction

A *Real-Time Execution Decision with Operational Distinction* is a tuple $\langle t$, *event-noops*$\rangle$, where:

- t is a time with domain $\mathbb{R}$,
- *event-noops* is a set of *event-noop* pairs to be executed at time t.

For convenience and simplicity, and given the similarities between RTED and RTED with Operational Distinction, future references to RTEDs will always refer to RTEDs with Operational Distinctions.

6.1.2 Dispatching Actions Dynamically

The dynamic dispatcher runs the main loop of the executive's temporal reasoning routine. It consists of a dispatching routine and some type of outer loop monitoring it.

The dispatching routine, Algorithm 6, is responsible for retrieving the latest RTEDs and dispatching actions when the clock indicates that the agent has reached time t corresponding to the latest RTED. The outer loop allows the dispatching routine to run until the scheduler reports there are no requirement events remaining.

We now provide a walkthrough of the dynamic dispatching algorithm. For simplicity's sake, the term *schedule* here is shorthand for whatever data structures the scheduler uses to generate RTEDs. *Updating the schedule* refers to running the fixed-delay FAST-EX update, Algorithm 3, using the variable-delay execution strategy from Section 5.2.

The interaction between the dispatching routine and monitoring loop is limited. Algorithm 6 returns a Boolean indicating whether there are executable events remaining. Here, the monitoring loop, Algorithm 4 is a simple `while` that repeats until it receives `false` from the inner loop.

We break the dispatching routine into three distinct phases.

1. Observe events that were executed.
2. Collect events from the RTED and events that have been buffered that should be dispatched at this time.
3. If there are events to be dispatched:
   (a) send real events to the driver, and
   (b) immediately assign all *no-op* events to the current time.

Our goal in the dispatching routine is to dispatch events to the driver only after updating the schedule, collecting an up-to-date RTED, checking for buffered events, and confirming we are within the time window of the actions to be dispatched. The routine will exit before reaching the dispatch step if any conditions are not met.

For the first step, we ask the scheduler if there are any remaining executable events. If there are none, we return `false` to signal the loop's termination, otherwise we continue.

Next, we observe events associated with actions that have been dispatched by the driver. We choose to use a FIFO queue to store messages corresponding to event

observations from the driver. The presence of a message would indicate that the driver has successfully executed a free event. We iteratively pop messages off the queue and update the schedule with the events and execution time contained in each message. Note that the scheduler update is a blocking operation because we need an up-to-date schedule to guarantee future RTEDs are consistent.

The second step begins once we have popped all messages from the driver off the queue. We need to decide what events will be dispatched as actions next. Given the relationship between the scheduler, routine, and driver, we do not assume that dispatched actions are executed instantaneously by the driver. We know that execution contends against delays such as the computational time in simply calling a function, to network latency, to robotic hardware that takes a moment to interpolate a motion plan from waypoints. In some contexts, it may make sense to preempt execution by dispatching events some small amount of time *before* the clock time reaches the RTED execution window. We call this preemption time ϵ, where $\epsilon \in \mathbb{R}^{\geq 0}$. If $\epsilon = 0$, the dispatcher is not allowed to preemptively dispatch actions before the RTED time. The value of ϵ should be dependent on the operational domain of the driver.

What distinguishes a delay dispatcher from a dispatcher that would work with instantaneous observations is a delay dispatcher's handling of buffered events (see Lemma 11). Given we left no facility in the delay scheduler to track events that need to be buffered, a delay dispatcher must take responsibility for buffered events. A delay dispatcher should record which events need to be buffered, and how long they should be buffered.

Algorithm 5 provides a subroutine in which we compare a new RTED and any buffered events to the history of actions dispatched so far. Let t be the current time. It outputs a set of event-noopsconsisting of:

1. any event buffered to t,
2. real events of the RTED if $t_{RTED} = t \leq \epsilon$, and
3. no-op events of the RTED if $t_{RTED} = t$.

Buffered events are always no-ops given that they are uncontrollable events ob-

served earlier, hence we do not need to preempt them with ϵ. No-op events in the RTED should not be preempted either. Only real events should be preempted.

A history of actions is necessary to avoid dispatching the same action more than once. If the dispatcher loop is running quickly and actions are dispatched asynchronously, then the loop may iterate one or more times between dispatching an action and observing its associated event.

If Algorithm 5 returns no *event-noop* pairs, we end this iteration by returning `true`.

Once we reach the third stage, we are guaranteed to be able to dispatch valid actions because (1) we have confirmed that the *event-noops* we have in hand have never been dispatched, and (2) we are in a time window that the scheduler has told us is consistent with the STNU's constraints. We filter the *event-noop* pairs into a set of no-op events and a set of real events. In the event that an uncontrollable event and its normalized lower bound (both no-ops) are to be scheduled at the same time, we schedule the normalized lower bound first first. Real events are then asynchronously sent to the driver.

All *event-noops* that were dispatched are added to the history to prevent them from being dispatched again. Finally, because events were dispatched, the dispatching routine returns `true`.

We benefit greatly from using instance-based properties. The implementation of the delay dispatcher for this thesis uses slots on a `dispatcher` class to manage inputs to the dispatcher, which are then accessed by reference. All inputs to Algorithm 4 can be properties on an instance of a delay dispatcher class.

Input:
Initialization: Hash-table `buffered-events` $\leftarrow \varnothing$; Set `history` $\leftarrow \varnothing$
Dynamic Dispatching Outer Loop:
1 `all-inputs` $\leftarrow\langle$`buffered-events`, `history`, *Scheduler*, *Driver*, *Queue*, *Clock*, $\epsilon\rangle$
2 **while** *Calling inner loop with* ***`all-inputs`*** *returns true* **do**
3 **continue**
4 **end**

Algorithm 4: The outer loop of the dynamic dispatching algorithm.

Input: `RTED`; `buffered-events`; a set of dispatched event-noops `history`; ϵ; current time t
Output: Set of event-noops
Initialization: `event-noops` $\leftarrow\varnothing$;
Choose Event-Noops:
if *t is a key in* `buffered-events` **then**
  `event-noops` $\leftarrow$ event-noops from `buffered-events`$[t]$;
endif
if `RTED`$[time] = t$ **then**
  Add no-op `RTED`[event-noops] to `event-noops`;
endif
if $t -$ `RTED`$[time] \leq \epsilon$ **then**
  Add real `RTED`[event-noops] to `event-noops`;
endif
Remove any event-noops in `event-noops` that are in `history`;
`return event-noops`;

Algorithm 5: An algorithm for paring the events from an RTED and buffered events into event-noops.

The biggest factor for the performance of the dispatching routine, Algorithm 6, is updating the schedule. Assuming the *Scheduler* is the Delay Scheduler described in Section 6.1, then performing an assignment of an event will trigger the FAST-EX update that runs in $O(N^3)$ [15, p. 144] with the number of events in the STNU. In the worst case, the dispatcher confirms that all events in the STNU have arrived at the same time, whether as messages from the driver in the FIFO queue, or RTED `noop` events. Each event would trigger a schedule update. Thus, the dynamic dispatching routine runs in $O(N^4)$ in the worst case.

6.1.3 Observing Contingent Events

The dispatcher relays contingent event observations to the scheduler. In the base case, when a contingent event is observed, the dispatcher updates the schedule with the event and current clock time.

If the observed event is uncontrollable and arrived earlier than its lower bound, then the dispatcher will save the event in a `buffered-events` hash-table with the lower bound of its constraint as the key. By Lemma 11, the lower bound will be $l^+(x_c)$ for some uncontrollable event x_c.

Input: Current time t,
`buffered-events`; `history`; *Scheduler*; *Driver*; *Queue*; *Clock*; ϵ;
Output: Boolean whether the outer loop should continue
Initialization: `real-events` $\leftarrow \{\}$; `noop-events` $\leftarrow \{\}$; $t \leftarrow$current time of *Clock*;

Dynamic Dispatching Routine:

```
1  if Scheduler has no more unexecuted events then
2  |   return false;
3  endif
4  for message in Queue do
5  |   Pop message;
6  |   for event, t_execution in message do
7  |   |   Update Scheduler with observation of event at t_execution;
8  |   end
9  end
10 RTED ← a new RTED from Scheduler; //Equations 5.2 and 5.4
11 event-noops ←choose-event-noops(RTED, buffered-events,
    history, ϵ, t);
12 if no event-noops then
13 |   return true;
14 endif
15 for event-noop pair in event-noops do
16 |   if event-noop[noop] is true then
17 |   |   Add event-noop[event] to noop-events;
18 |   else
19 |   |   Add event-noop[event] to real-events;
20 |   endif
21 end
22 Sort noop-events such that normalized lower bounds have the lowest
    indices;
23 for event in noop-events do
24 |   Update Scheduler with observation of event at t;
25 end
26 Asynchronously send all real-events to the Driver;
27 Add event-noops to history;
28 return true;
```

Algorithm 6: The dynamic dispatching routine.

6.1.4 Experimental

Finally, we benchmark action dispatching. In our simulated environments for dispatching, we run the dispatcher function as described in Algorithm 6 twice per simulated second. (We run it twice in the event that scheduling an event enables us to dispatch other actions immediately. If we ran Algorithm 6 once per second, the newly enabled events would then be dispatched a second late.)

Given every event will be scheduled once using the FAST-EX update, FAST-EX updates will dominate the total runtime of dispatching. As seen in Figure 6-2, the total runtime of all calls to Algorithm 6 indeed follows $O(N^2 \log N)$.

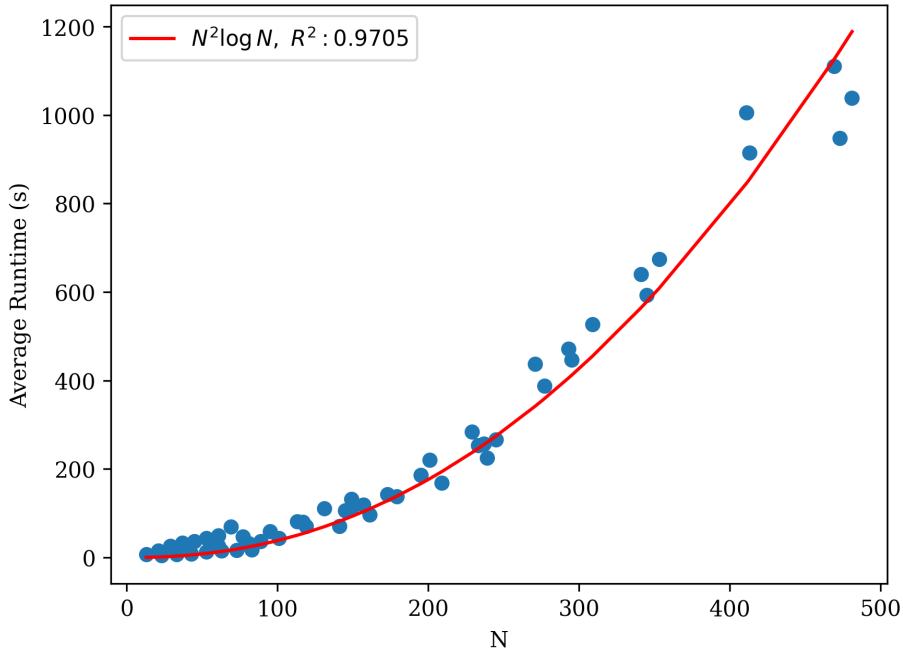


Figure 6-2: Average runtime data for running Algorithm 6.

6.2 Architecture

We present a view of the Delay Kirk architecture that focuses attention to its scheduling and dispatching capabilities. Kirk takes RMPL [16] as input and produces actions as output (from here on, "Kirk" refers to Delay Kirk because the architectural design

of Delay Kirk and other Kirks is fundamentally the same). As shown in Figure 6-3, there are three key components of Kirk.

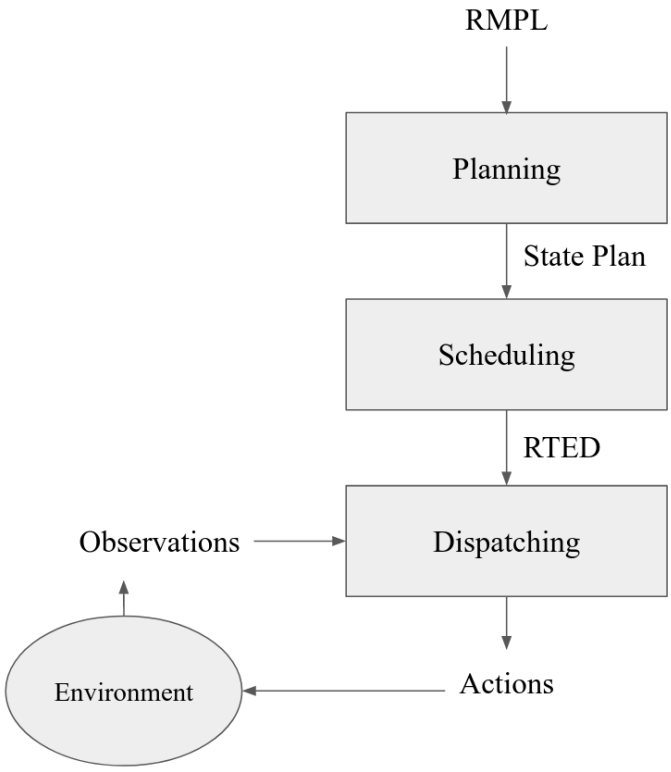


Figure 6-3: A simplified, high-level overview of the Delay Kirk task executive architecture with respect to dispatching actions.

Figure 6-3 explicitly identifies the environment. We do so to highlight that Kirk is designed to be able to interact with the outside world. For instance, if Kirk is running on a robot, the environment might consist of the pose of the manipulator and any objects in the scene. If Kirk is responsible for sending notifications to a digital assistant in a spacesuit, then the environment might be the "as executed" version of an EVA timeline. In either case, actions caused by Kirk will impact the environment. Likewise, Kirk learns from the environment. Here we show event observations from the environment being sent to the scheduler. However, when Kirk is working with sub-executives designed for specific problem domains, e.g. risk-bounded motion planning, it may be monitoring other aspects of the environment as well.

Every Kirk has a planning component that takes RMPL as input, generates state plans, then checks consistency using OpSAT. OpSAT is similar to a satisfiability

(SAT) solver with the property that it produces optimal assignment to real valued variables. Any temporal constraints in the state plan are translated to a delay STNU then checked with the variable-delay controllability checker from Chapter 4.

If the overall state plan is satisfiable, it is then sent to the delay scheduler. Note that earlier we have said that the delay scheduler takes a temporal network as input. However, Figure 6-3 shows a state plan as input to the delay scheduler. Functionally, there is no difference. There is a one-to-one relationship between state plans and delay STNUs. In fact, as implemented for this thesis, the delay scheduler can take either a state plan or delay STNU as input. If a state plan is received, then the first action taken is to convert the state plan to a delay STNU.

RTEDs that the delay scheduler outputs are sent to a delay dispatcher.

6.3 RMPL

RMPL [38] is a key component of Kirk. This section steps through example RMPL control programs to describe their features and our modeling choices. The purpose of this section is two-fold:

1. We must describe the modeling choices of RMPL in sufficient detail to make concrete our approach to modeling temporal constraints in human-readble form for the experiments in Sections 7.3.1 and 7.3.2
2. The above is used to demonstrate that modeling uncertain communication delay can be naturally modeled in RMPL.

This section is not meant to be a complete documentation of RMPL, rather our goal is to motivate the strength of RMPL as a modeling language for human planners describing autonomous systems with observation uncertainty.

RMPL has undergone a number of rewrites since its inception, and is currently being developed as a superset of the Common Lisp language using the Metaobject Protocol [41]. The goal is that a human should have a comfortable means for accurately modeling sufficient detail about the problem domain such that an executive can perform model-based reasoning to decide how to act.

An example of an RMPL control program for a single-agent without agent dynamics follows in Listing 1.

```
;; NOTE: we omitted Lisp package definitions here for simplicity's sake

(define-control-program eat-breakfast ()
  (declare (primitive)
           (duration (simple :lower-bound 15 :upper-bound 20))))

(define-control-program bike-to-lecture ()
  (declare (primitive)
           (duration (simple :lower-bound 15 :upper-bound 20))))

(define-control-program main ()
  (with-temporal-constraint (simple-temporal :upper-bound 40)
    (sequence (:slack nil)
              (eat-breakfast)
              (bike-to-lecture))))
```

Listing 1: A sample control program composed of three constraints. `eat-breakfast` and `bike-to-lecture` designate controllable constraints, while the `main` control program enforces that the constraints are satisfied in series.

Looking past the parentheses, we can see different options for defining temporal constraints. For example, the `(duration (simple ...))` form is used to define a set-bounded temporal constraint between a `:lower-bound` and an `:upper-bound`. The `main` control program uses a different form, `(with-temporal-constraint ...)` to place an `:upper-bound` on the overall deadline for scheduling all events in the control program.

The example control programs in Listing 1 are defined without agents in that there is an assumption that the Kirk instance that executes this control program must know what the semantics of `eat-breakfast` and `bike-to-lecture` mean and how to execute them.

Each constraint is represented as an episode of a start and end event, e.g. `eat-breakfast` becomes `eat-breakfast:start` $\xrightarrow{[15,20]}$ `eat-breakfast:end`.

It could also be the case that Kirk is simply being used to produce a schedule of events offline that will be handed to an agent that knows how to execute them.

As an example, perhaps a student wants some help planning their morning, so they write an RMPL control program with constraints representing everything they need to do between waking up and going to lecture, as seen in the more complex control program in Listing 2. The student could ask Kirk to produce a schedule of events that satisfies all the temporal constraints in this RMPL control program, which they would then use to plan their morning routine. See the resulting schedule produced by Kirk in Table 6.1. (Note that while normally times in RMPL are represented in seconds, we use minutes in Listing 2 and Table 6.1 for simplicity's sake.)

Table 6.1: The schedule produced by Kirk's scheduler for the student's routine before lecture as modeled in Listing 2. Note: Kirk's output has been cleaned for readability purposes.

Event	**Time (min)**
`START`	0
Start `shower`	1
End `shower`	6
Start `review-scheduling-notes`	6
Start `eat-breakfast`	6
End `review-scheduling-notes`	16
Start `review-planning-notes`	16
End `eat-breakfast`	21
End `review-planning-notes`	26
Start `pack-bag`	26
End `pack-bag`	31
Start `bike-to-lecture`	32
End `bike-to-lecture`	46
`END`	46

Listing 2 introduces the notion of control programs that are allowed to be executed simultaneously, as modeled with the `(parallel ...)` form found in the `main` control program on line 48.

Kirk is able to simulate the RMPL script in Listing 2 and produce a schedule because there were no uncontrollable constraints, that is, all control programs are under the agent's control. Say we replaced `bike-to-lecture` with `drive-to-lecture`. Due to traffic conditions, driving presents in an uncontrollable constraint. RMPL allows us to model uncontrollable constraints as in Listing 3.

The addition of `:contingent t` to the `(duration ...)` form tells Kirk that

```
;; This file lives in the thesis code repo at:
;;      kirk-v2/examples/morning-lecture/script.rmpl
;;
;; To execute this RMPL control program as-is and generate a schedule, go to the root
;; of the thesis code repo and run the following command:
;;
;; kirk run kirk-v2/examples/morning-lecture/script.rmpl \
;;      -P morning-lecture \
;;      --simulate

(rmpl/lang:defpackage #:morning-lecture)

(in-package #:morning-lecture)

(define-control-program shower ()
  (declare (primitive)
           (duration (simple :lower-bound 5 :upper-bound 10))))

(define-control-program eat-breakfast ()
  (declare (primitive)
           (duration (simple :lower-bound 15 :upper-bound 20))))

(define-control-program review-scheduling-notes ()
  (declare (primitive)
           (duration (simple :lower-bound 10 :upper-bound 15))))

(define-control-program review-planning-notes ()
  (declare (primitive)
           (duration (simple :lower-bound 10 :upper-bound 15))))

(define-control-program pack-bag ()
  (declare (primitive)
           (duration (simple :lower-bound 5 :upper-bound 6))))

(define-control-program bike-to-lecture ()
  (declare (primitive)
           (duration (simple :lower-bound 15 :upper-bound 20))))

(define-control-program review-notes ()
  (sequence (:slack t)
    (review-scheduling-notes)
    (review-planning-notes)))

(define-control-program main ()
  (with-temporal-constraint (simple-temporal :upper-bound 60)
    (sequence (:slack t)
      (shower)
      (parallel (:slack t)
        (eat-breakfast)
        (review-notes))
```

```
(define-control-program drive-to-lecture ()
  (declare (primitive)
           (duration (simple :lower-bound 15 :upper-bound 20)
                     :contingent t)))
```

Listing 3: An uncontrollable, or contingent, temporal constraint in a control program.

`drive-to-lecture:end` is an uncontrollable event. With the instantiation of Kirk used for this thesis, observations of `drive-to-lecture:end` could come in the form of user interactions, HTTP POST requests, or a pre-determined list of event observations given to Kirk.

As a contribution of this thesis, our existing approach to specifying durations in RMPL was expanded to model observation delay. An example follows in Listing 4 modeling a sample collection control program with observation delay.

```
(define-control-program collect-science-sample ()
  (declare (primitive)
           (duration (simple :lower-bound 15 :upper-bound 30
                             :min-observation-delay 5
                             :max-observation-delay 15)
                     :contingent t)))
```

Listing 4: An RMPL control program describing a science data collection task with observation delay.

We can see in Listing 4 that representing set-bounded observation delay is a simple as adding `:min-` and `:max-observation-delay` to the `(duration (simple ...) :contingent t)` form.

See Appendix B for further discussion of RMPL.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 7

Coordinating Multiple Agents under Uncertain Communication

In this chapter, we present a novel MA framework for dynamic event scheduling with inter-agent temporal constraints. Our framework adheres to the variable observation delay modeling framework presented in Chapter 4, making it robust to uncertain communication.

Online MA coordination of event dispatching allows executives to dynamically decide when to act given the resolution of inter- and intra-agent temporal constraints. In our formulation, each executive has its own STNU with contingent events it expects to observe and free events it is responsible for monitoring. We do not distinguish between contingent events that are the free events scheduled by peer agents and contingent events from any other source in Nature. There are no restrictions on inter-agent constraints, though they must avoid chained contingencies the same way that vanilla, single-agent STNUs do [27].

Executives are *not* required to have perfect knowledge of the complete state of the world with respect to event assignments, nor are they required to even *agree* on the state of the world. Rather, their knowledge should be consistent with the temporal constraints and observation delay modeled in their individual delay STNUs. This requirement stands in opposition to the communication challenge that is commonly addressed in problems involving distributed agents, e.g. distributed consensus

approaches [42], [43], where it is crucial that all agents agree on the state of the world. In our framework, each agent acts according to their given constraints. Due to scheduling uncertainty from observation delay, it is impossible to expect agents to agree on partial histories, nor is it necessary because each agent is capable of scheduling events with observation uncertainty.

To our knowledge, no such online scheduler for MA coordination with this requirement has been proposed. In this chapter, we first present a grounded Artemis-like scenario to motivate coordination. Next, we describe a modeling framework for MA control programs that is necessary for establishing coordination between agents. Then we define an event propagation algorithm used to guarantee that event observations match individual agent STNUs. We finish by presenting experimental analysis of our event propagation algorithms, and the results of hardware demonstrations of Delay Kirk using a robotic arm and a simulated astronaut.

7.1 Event Propagation

At a high level, scheduled events propagate through a simple directed graph of connected executives. We put checks in place to ensure that cycles do not cause infinitely recursed event observations.

Definition 30. Communication Graph

A *communication graph* C is a tuple $\langle V, E \rangle$, where:

- V is a set of vertices representing peer executives,
- E is a set of directed edges between $v \in V$ representing the path of event observation propagation,
- Each edge $e_i \in E$ is a pair (o, t), where $o, t \in V$ represent the origin and termination of the edge respectively.

Self-loops, or self-edges, are not allowed, i.e. for any vertex $v_i \in V$, no single edge $e_i \in E$ may both originate and terminate at v_i.

For some executive $v_i \in V$ with outgoing edges in E, (v_i, v_j), $\cdots$, (v_i, v_k), any scheduled events that v_i assigns, whether free or contingent, are propagated to all peer executives v_j, $\cdots$, v_k. Likewise, all contingent events received from Nature are propagated to peers. Finally, any events v_i receives from other agents are also relayed to peers.

Definition 31. Event Propagation Messages

An *event propagation message* m is a tuple $\langle x, P \rangle$, where:

- x is a set of one or more events scheduled simultaneously,
- $P \subseteq V$ is a set of executives who have already received the message.

Recognize that Definition 31 is vague in defining x. Event propagation messages are passed between agents, and each agent has its own STNU. In some cases, x will be free events, in others x will be contingent events. The type of event makes no difference to the algorithm so we do not distinguish between them here.

Events that are received in m, $m[x]$, are handled the same as observations of contingent events during scheduling. Lemmas 14, 17, and 16 are applied as appropriate when the observation of $m[x]$ arrives.

For an edge $(v_i, v_j) \in E$, it is possible that v_j receives events that are not present in its STNU.

Because we have not defined a temporal decoupling-like algorithm wherein an STNU for multiple-agents is programmatically separated into individual STNUs (see the discussion of multi-agent STNUs [11] in Section 8.1), we are reliant on human planners to write STNUs for each agent by hand. As a result, there is no guarantee that x is meaningful to a given agent.

To be more specific, there is no guarantee that any event $x_i \in x$ in the event propagation message has an equivalent event in X_c of the STNU being executed by any receiving agent $v_j \in V$. If agent v_j cannot find x_i in their X_c, then x_i can be ignored. As will be discussed in Algorithm 7, we represent x using a type that can be compared for equivalence with the events in an agent's STNUs, e.g. a list of strings.

We use P to avoid cycles in event propagation. As will be shown in Algorithm 7, agent v_i will avoid propagating x to any agents in P. Agent v_i will also grow P when it relays m to other agents by appending to P itself and all outgoing agents $v_j, \cdots, v_k$.

Timing information, e.g. timestamps, is explicitly excluded from m. Dynamic scheduling and the variable-delay STNU and event observation, `obs`, formalisms do not account for timestamps. Instead, we expect that passing messages for event propagation between executives takes an amount of time in the domain $\mathbb{R}^+$. Thus, when v_j expects to receives an event, $x_i \in x$, from v_i, the time delay can be naturally modeled in the variable-delay function, $\bar{\gamma}(x_i)$, in the STNU that v_j will execute.

If event propagation messages were to include accurate timestamps, we would need to modify the way events are recorded during scheduling, impacting scheduling Lemmas 14, 17, and 16. Scheduling events in the past could also impact controllability. For these reasons, we avoid the inclusion of timestamps in event propagation messages.

By Definition 31, events received from other agents are no different than events received from Nature, and no special considerations are required for scheduling.

We now walk through the process of passing messages between agents as shown in Algorithm 7. We use the same *Event Propagation* algorithm in three cases:

1. When an agent v_i schedules free events x,
2. When v_i receives an observation from Nature of contingent events x,
3. When v_i receives an incoming message m_i with contingent events $m_i[x]$ from another agent in V.

Let `peers` be a mutable set initialized to the terminal vertices for all $e \in E$ originating at v_i.

In the first case, agent v_i fulfills its responsibilities as defined in C by broadcasting x to its `peers`, who will receive x as exogenous contingent events. The outgoing message m_o that will be passed to `peers` will include enough information such that no agent should receive a given x more than once. To do so, we let P be a set of

all agents that will have observed x when m_o is received by `peers`, $P = \{v_i, p \; \forall \; p \in$ `peers`$\}$. We finalize $m_o = \langle x, P \rangle$, which we simultaneously transmit to each p in `peers`. Transmission is a "fire and forget" operation, where v_i does not wait for acknowledgment from any p that m_o was received.

The second case plays out the same as the first, the only difference being that x is itself observed from Nature. Once again, we let P be a list of v_i and all `peers`, and then transmit m_o simultaneously to all `peers`.

The third case is a relay operation. Agent v_i is responsible for propagating events $m_i[x]$ that it has just observed, but we want to avoid sending the events to `peers` who have already observed them. We remove those agents from `peers` accordingly with a set difference operation: `peers` = `peers` $- \; m_i[P]$. Likewise, we grow the list of agents who have received x, which is now $P = P \cup$ `peers`. Agent v_i composes a new $m_o = \langle m_i[x], P \rangle$ and transmits it to `peers`.

Ideally, the Event Propagation algorithm should run on a separate thread from the main scheduling loop, else we run the risk of incurring unnecessary delays in observing and dispatching events.

Input: Incoming message m_i; Scheduled events x; Self $v_i \in V$; Set of outgoing `peers` $\subset V$

Event Propagation:

1 `peers` $\leftarrow$ `peers` $- \; m_i[P]$;
2 $P \leftarrow m_i[P] \cup \{v_i\} \cup$ `peers`;
3 $x \leftarrow x$ or $m_i[x]$;
4 $m_o \leftarrow \langle x, P \rangle$;
5 **for** *each* p *in* `peers` **do**
6 | Perform a non-blocking transmission of m_o to p;
7 **end**

Algorithm 7: An event propagation algorithm that avoids recursive message passing.

The complexity of Algorithm 7 is trivially $O(N)$, where N is the number of executives in $V - 1$. The limiting factor to the performance of Event Propagation will be the time it takes to transmit messages between agents, which, to reiterate, should be modeled in the delay functions for any inter-agent temporal constraints.

7.2 Modeling Inter-Agent Constraints

Consider two agents, `agent1` and `agent2`, that are scheduling STNUs S_1 and S_2 respectively. S_1 and S_2 share a subset of semantically similar episodes, e_1 and e_2. `agent1` "owns" e_1, meaning it is responsible for scheduling the free event e_1-start and observing the contingent event e_1-end, while `agent2` owns e_2. It is the case that e_1 must precede e_2 in S_1 and S_2. A simplified MA view of the constraints is as follows.

$$e_1\text{-start} \overset{[15,30]}{\Longrightarrow} e_1\text{-end} \overset{[0,\infty]}{\longrightarrow} e_2\text{-start} \overset{[22,26]}{\Longrightarrow} e_2\text{-end}$$

From `agent1`'s perspective, S_1 models the following constraints. We add a `noop` start event, Z, to simplify coordination. For now, we allow chained contingencies, though in a moment they will need to be addressed.

$$Z \overset{[0,0]}{\longrightarrow} e_1\text{-start} \overset{[15,30]}{\Longrightarrow} e_1\text{-end} \overset{[0,\infty]}{\Longrightarrow} e_2\text{-start} \overset{[22,26]}{\Longrightarrow} e_2\text{-end}$$

We assume that `agent1` models e_2 in S_1 because other events under their control depend on e_2. S_2 is then modeled as follows. Note the change to the controllability of $Z \overset{[0,0]}{\Longrightarrow} e_1$-start.

$$Z \overset{[0,0]}{\Longrightarrow} e_1\text{-start} \overset{[15,30]}{\Longrightarrow} e_1\text{-end} \overset{[0,\infty]}{\longrightarrow} e_2\text{-start} \overset{[22,26]}{\Longrightarrow} e_2\text{-end}$$

For the sake of controllability of S_1 and S_2, we would simply add $[0, 0]$ free constraints between consecutive contingent constraints. Also note that from a scheduling standpoint, there is no difference between $a \overset{[0,0]}{\longrightarrow} b$ and $a \overset{[0,0]}{\Longrightarrow} b$ - both indicate a and b should be scheduled simultaneously.

We will walk through scheduling this scenario from the perspective of both agents. First, we describe their actions in the case that there is no communication delay, then we introduce communication, and finally we add delay to communications. This scenario will motivate our analysis of the challenges that arise in MA control programs.

If both agents have perfect knowledge of the world (instantaneous knowledge of events), scheduling is trivial. `agent1` and `agent2` execute Z simultaneously. `agent1`

schedules e_1-start and `agent2` instantaneously receives an observation of e_1-start. e_1-end arrives in $[15, 30]$ later, which again, both agents observe simultaneously. Now `agent2` is free to act. It schedules e_2-start, which `agent1` observes instantaneously. e_2-end arrives $[22, 26]$ later and is observed simultaneously by both agents.

Now, we enforce that `agent1` "owns" e_1 and is the only agent that can observe it directly. Likewise, `agent2` owns e_2. In order for an agent to learn about an episode they do not own, they must receive a communication from the agent who does. After `agent1` schedules e_1-start, it must send a message to `agent2`. `agent2` receives said message, which it interprets as an observation of e_1-start. If communications are instantaneous, the partial histories of both agents agree on the assignment of e_1-start. Later e_1-end is observed by `agent1`, who is then responsible for relaying a communication to `agent2` indicating that it is safe to assign e_1-end. `agent2` is now free to schedule e_2-start, which it does instantaneously. The same pattern of sending messages that events have been scheduled repeats and `agent1` learns that e_2-start was schedule simultaneously with e_1-end. After all events have been scheduled, the histories of `agent1` and `agent2` still agree on the times assigned to each event.

We now show that adding delay to the communications between agents forces us to add *synchronization* episodes to S_1 and S_2 to maintain event ownership. First, we must address the chained contingencies. Note that we have freedom in how we model the constraints of this scenario. The following example will motivate the need for a synchronization episode while remaining as close to the semantics of the original STNU as possible.

From the perspective of `agent1`, S_1, we cannot escape the fact that there are two uncontrollable events in sequence - the end of e_1 and the start of e_2, if we try to separate the events with a synthetic requirement episode, σ, with a $[0, \infty]$ constraint, the semantics no longer respect the original scenario.

$$Z \xrightarrow{[0,0]} e_1\text{-start} \xRightarrow{[15,30]} e_1\text{-end} \xrightarrow{[0,0]} \sigma\text{-start} \xrightarrow{[0,\infty]} \sigma\text{-end} \xRightarrow{[0,0]} e_2\text{-start} \xRightarrow{[22,26]} e_2\text{-end}$$

The delay scheduler will choose to schedule σ-end simultaneously with σ-start, also leading to e_2-start being immediately scheduled. However, e_2 is not under `agent1`'s control, and thus it has no authority to schedule e_2-start. Instead, our synthetic constraint also needs to be contingent.

$$Z \xrightarrow{[0,0]} e_1\text{-start} \xRightarrow{[15,30]} e_1\text{-end} \xrightarrow{[0,0]} \sigma\text{-start} \xRightarrow{[0,\infty]} \sigma\text{-end} \xrightarrow{[0,0]} e_2\text{-start} \xRightarrow{[22,26]} e_2\text{-end}$$

Now, the issue is that S_1 is uncontrollable due to σ-`start` $\xRightarrow{[0,\infty]}$ σ-`end`. We know the `agent2` will receive e_1-end somewhere in $\bar{\gamma}'(e_1\text{-}\texttt{end})$, where the $\bar{\gamma}'$ function represents observation delay in S_2. `agent2` will then immediately schedule e_2-start. Finally, S_1 becomes

$$Z \xrightarrow{[0,0]} e_1\text{-start} \xRightarrow{[15,30]} e_1\text{-end} \xrightarrow{[0,0]} \sigma\text{-start} \xRightarrow{[\bar{\gamma}'^{-}(e_1\text{-}\texttt{start}),\bar{\gamma}'^{-}(e_1\text{-}\texttt{end})]} \sigma\text{-end} \xrightarrow{[0,0]} e_2\text{-start} \xRightarrow{[22,26]} e_2\text{-end}$$

In practice, an agent may choose to schedule other events while waiting for σ-end to arrive.

In S_2, we may choose to give `agent2` the same synchronization episode without changing the execution semantics. We know that e_1-end will be observed somewhere in $\bar{\gamma}'(e_1\text{-}\texttt{end})$. When e_1-end arrives, we are guaranteed to have waited somewhere in the lower and upper bounds σ. Assuming `agent2` knows that e_1-end and σ-end semantically represent the same point in time, σ-end can be safely scheduled as soon as e_1-end arrives.

Synchronization episodes allow inter-agent constraints with observation delay to be modeled without impacting the ordering of events. They are used to separate control programs in the hardware demonstration in Section 7.3.2.

7.3 Experimental Analysis

We performed two demonstrations of the Event Propagation algorithm. The first was a hardware demonstration performed on a Barrett WAM manipulator in the MERS lab. The second is a multi-agent simulation showcasing inter-agent constraints. Both will be described below.

7.3.1 Distributed Kirk Simulation

To demonstrate multi-agent communication, we built a simulation of an end-to-end mission with three independent Kirks, `agent0`, `agent1`, and `agent2`. We will show that distributed Kirks can successfully dispatch events within temporal bounds in the face of multiple sources of communication uncertainty. The Kirks are responsible for executing an installation procedure with the same randomly generated constraints as used in the validation of the delay scheduler in Section 5.3. In this scenario, each agent is responsible for installing two satellite dishes with staggered confirmations so as to limit uplink bandwidth usage. As Kirks receive confirmation that installation has been completed, they then share the confirmations with their peers.

To simplify comparing schedules, we used a standardized format for event names. Repeated event names are given as `Event:[agent]:[iteration]`, where `[agent]` and `[iteration]` are zero-indexed. For instance, `Install:4:3` would be the start of an installation episode for a hypothetical `agent4` (of at least five agents) in its fourth iteration.

There is one modification from the original constraints from Section 5.3 in that we separate the communication delay inherent to the confirmation task with the observation delay inherent to sharing observations with peers. There may be a delay waiting for confirmation from ground, and the *in situ* communication infrastructure may add an additional delay to communications between agents. We assume the sources of delay compound. For instance, `agent1` will need to know when `agent0` has confirmed its installation task, `Confirm:0:0` before beginning their own installation, `Install:1:0`. if `agent0` expects to receive `Confirm:0:0` with an observation delay

of $\bar{\gamma}(\texttt{Confirm:0:0}) = [0, 10]$, we increase $\bar{\gamma}^{+}(\texttt{Confirm:0:0})$ by one for any peers that receive the observation broadcasted from `agent0`. In other words, from the perspective of `agent1` or `agent2`, $\bar{\gamma}(\texttt{Confirm:0:0}) = [0, 11]$ instead.

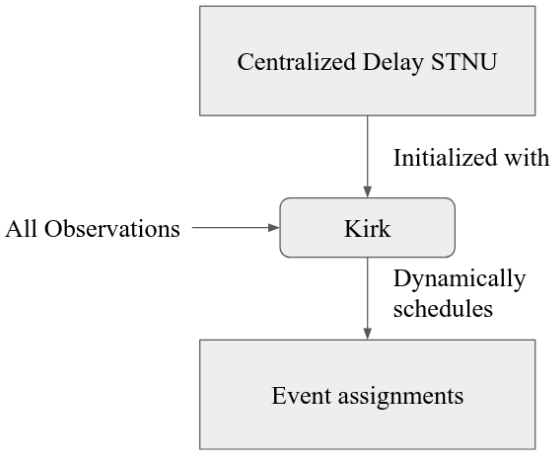


Figure 7-1: The Kirk architecture used to generate event assignments for the centralized delay STNU. A single Kirk receives the VDC STNU that includes constraints for all agents, as well as contingent event observations. Kirk then performs delay scheduling, resulting in an assignment to all events.

At a high-level, our procedure for creating this demonstration is as follows. We randomly generated a variable-delay STNU for three agents and two installation procedures (using the same generator code that was used in Section 5.3) and confirmed it to be VDC. We call this STNU the *centralized delay STNU* in that it includes all constraints for all three agents in a multi-agent mission with observation delay. We then acted like a mission planner in that we manually decoupled the centralized delay STNU into three single-agent RMPL control programs. Each control program contained the subset of the constraints from the centralized delay STNU required for a single agent to maintain the semantics of the original constraints. We call the variable-delay STNUs represented by the collection of the three RMPL control programs the *distributed variable-delay STNUs*. We finally pre-determined when observations would arrive for each agent to simplify running the demonstration. Both the centralized and distributed scenarios received observations of the same events at the same times.

The architecture for the centralized scenario is shown in Figure 7-1, while the

distributed scenario is represented in Figure 7-2. Figure 7-2 presents a simplified view in order to keep the diagram readable. In reality, each Kirk broadcasts all events to all peers.

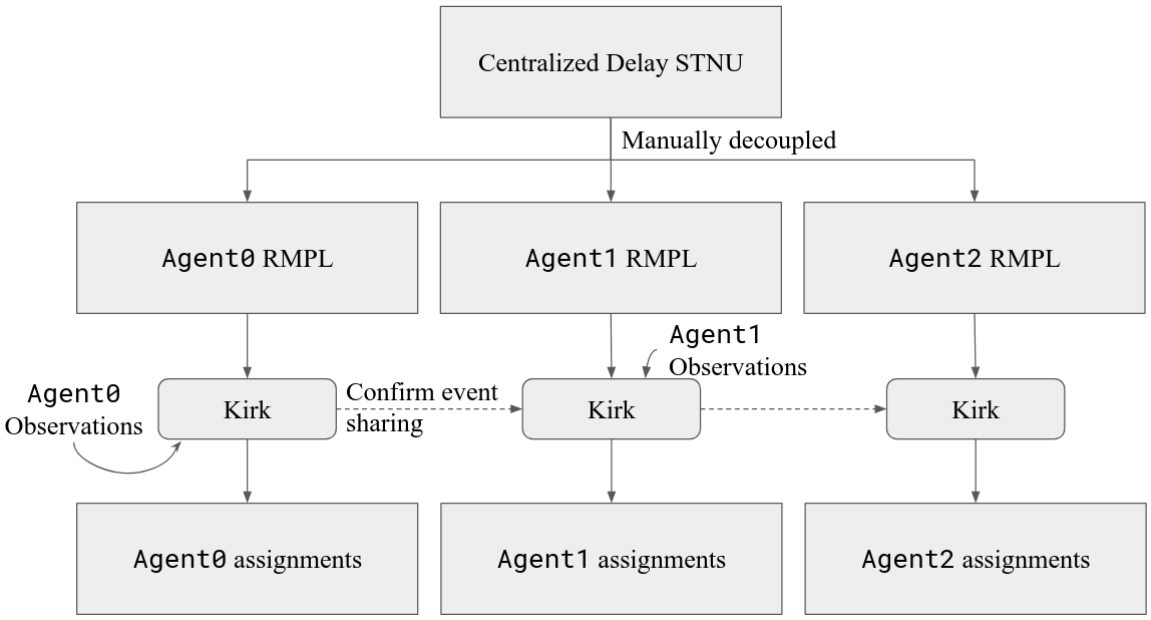


Figure 7-2: The distributed architecture for the demonstration. The original centralized delay STNU is manually decoupled to three separate RMPL control programs, which are then used to initialize three Kirks. The Kirks receive appropriate event observations, which they then share to their peers. After delay scheduling, each Kirk produces an assignment to events that were under their control.

Event observations were arranged as follows. In the centralized case, the single Kirk received all contingent event observations. Any observations that were not explicitly provided as an observation was assumed to be assigned at its upper bound. In the distributed case, Kirks were only given event observations for events that belong to them. For instance, only `agent0` received an observation of `Confirm:0:0`, the event signifying that they have completed installation of the first satellite. It was then the responsibility of `agent0` to broadcast the event observation to its peers.

To evaluate the ability of a distributed Kirk architecture to perform scheduling with communication uncertainty, we focus on the schedules produced. To do so, we compare the schedule created by a Kirk running against centralized delay STNU

(Table 7.1) against the combined schedules of the three single agents (Tables 7.2-7.4). If observations arrive at the same time, both scenarios should yield the same schedules. Importantly, the inter-agent constraint between overlapping installation tasks should hold in the distributed scenario. The confirmation events are highlighted in gray in each table for ease of identification.

Running the demonstration was then a matter of running three networked instances of Kirk simultaneously against three different control programs. We did so using a `Makefile` with three targets, running `make` with the `-j3` flag, and setting up communications to take place over HTTP.

From the root of the thesis repository, execute `make kirk && make -j3 demo` to run the demonstration. The resulting schedules will be written to `agent{0,1,2}.txt`. Note that the STNU was generated directly for the centralized delay STNU, but the STNUs were compiled from RMPL control programs for the distributed delay STNUs. There are naming differences between the events of the different schedules due to way control programs receive names in RMPL and the way RMPL control programs are compiled to STNUs. The event names in schedules in Tables 7.1-7.4 have been manually altered such that they match here. See Appendix B for a description of the resulting STNUs from RMPL control programs.

Here, we show `Confirm:0:1` as the last event, but In the RMPL control program, we used a `close-out` episode with bounds $[0, \infty]$ to end the mission. Given that it follows a `Confirm` episode, It is semantically the same as the confirmation (again, see Appendix B for an explanation of how control programs translate to STNUs).

We can see in Tables 7.1-7.4 that the three Kirks are able to avoid overlapping installation tasks using a communication architecture that assume uncertain communication.

7.3.2 Hardware Demonstration

We envision a scenario with an astronaut and a robot coordinating on the lunar surface. The astronaut is performing scientific exploration while the robot performs remote construction tasks. The concept of operations allows for the astronaut to use

Table 7.1: The schedule produced by a single Kirk against against the "multi-agent" variable-delay STNU.

Event	**Time (s)**
ALL:START	0
Start:0:0	0
Start:1:0	0
Normalized Lower for Traverse:0:0	1
Normalized Lower for Traverse:1:0	1
Traverse:0:0	10
Install:0:0	11
Normalized Lower for Confirm:0:0	15
Traverse:1:0	15
Confirm:0:0	17
Install:1:0	17
Start:2:0	17
Normalized Lower for Traverse:2:0	18
Start:0:1	19
Normalized Lower for Traverse:0:1	20
Normalized Lower for Confirm:1:0	22
Confirm:1:0	30
Start:1:1	31
Normalized Lower for Traverse:1:1	32
Traverse:2:0	32
Install:2:0	33
Traverse:0:1	38
Install:0:1	39
Normalized Lower for Confirm:2:0	40
Confirm:2:0	41
Normalized Lower for Confirm:0:1	42
Start:2:1	43
Normalized Lower for Traverse:2:1	44
Traverse:1:1	44
Install:1:1	45
Confirm:0:1	48
Normalized Lower for Confirm:1:1	52
Confirm:1:1	55
Traverse:2:1	60
Install:2:1	61
Normalized Lower for Confirm:2:1	62
Confirm:2:1	63
ALL:END	63

Table 7.2: The single agent schedule produced by `agent0` in the demonstration.

Event	**Time (s)**
Start:0:0	0
Normalized Lower for Traverse:0:0	1
Traverse:0:0	10
Install:0:0	11
Normalized Lower for Confirm:0:0	15
Confirm:0:0	17
Start:0:1	19
Normalized Lower for Traverse:0:1	20
Traverse:0:1	38
Install:0:1	39
Normalized Lower for Confirm:0:1	42
Confirm:0:1	48

Table 7.3: The single agent schedule produced by `agent1` in the d emonstration.

Event	*Time (s)
Start:1:0	0
Start:0:0	0
Normalized Lower for Confirm:0:0	6
Confirm:0:0	17
Traverse:1:0	17
Install:1:0	18
Normalized Lower for Confirm:1:0	23
Confirm:1:0	30
Start:1:1	31
Normalized Lower for Traverse:1:1	32
Traverse:1:1	44
Install:1:1	45
Normalized Lower for Confirm:1:1	52
Confirm:1:1	55

Table 7.4: The single agent schedule produced by `agent0` in the demonstration. We added a `CLOSE-OUT` episode to end with a requirement event.

Event	**Time (s)**
Start:2:0	17
Normalized Lower for Confirm:1:1	22
Confirm:1:1	30
Traverse:2:0	32
Install:2:0	33
Normalized Lower for Confirm:2:1	40
Confirm:2:1	41
Start:2:1	43
Normalized Lower for Traverse:2:1	44
Traverse:2:1	60
Install:2:1	61
Normalized Lower for Confirm:2:1	62
Confirm:2:1	63

a rover to traverse away from the robot in search of promising scientific samples. Due to the position of surface relays and general uncertainty in lunar topology, there is an uncertain time delay between agents.

Bandwidth between Mission Control on Earth and the Moon is limited. There are low and high bandwidth communications available to both agents. Low bandwidth is responsible for transmitting critical data (e.g. suit telemetry), while high bandwidth communications are reserved for purposes such as video calls and large dumps of scientific data. It is not possible for both the astronaut and the robot to use high bandwidth communications simultaneously. Thus, there is a need for the agents to coordinate such that they make effective use of high bandwidth communications without stepping on each others toes, so to speak.

We hone in on a point in an EVA where there is substantial time delay between the astronaut and robot. The astronaut has set out far from the robot in search of scientifically interesting rock samples. Meanwhile, the robot is preparing to perform a drilling operation. The astronaut's sample collection work involves spectroscopy and video imagery, which is being sent to Mission Control using the high bandwidth connection. It will take between 15 and 30 minutes to downlink all the data. As soon as sample collection is over, the robot can use the high bandwidth connection

to stream video back to scientists and engineers on earth while performing a drilling operation.

We say that the astronaut "owns," or is responsible for sharing observations of, the start and end of the experiment, while the robot similarly owns the drilling operation.

We built a physical demonstration of this scenario of this thesis in our laboratory using a Barrett WAM manipulator and a simulated astronaut. In this scenario, the astronaut and robot are collaborating on the lunar surface with uncertain communication delay between them. Throughout this mission, agents must coordinate with respect to their usage of uplink bandwidth. We choose to focus on a moment in time where the robot is waiting for the human to finish their use of the uplink before beginning a bandwidth-heavy task of their own. We assume the agents are moving on the lunar surface during execution, and as such the observation delay between them changes as well. See Figure 7-3 for the laboratory setup.

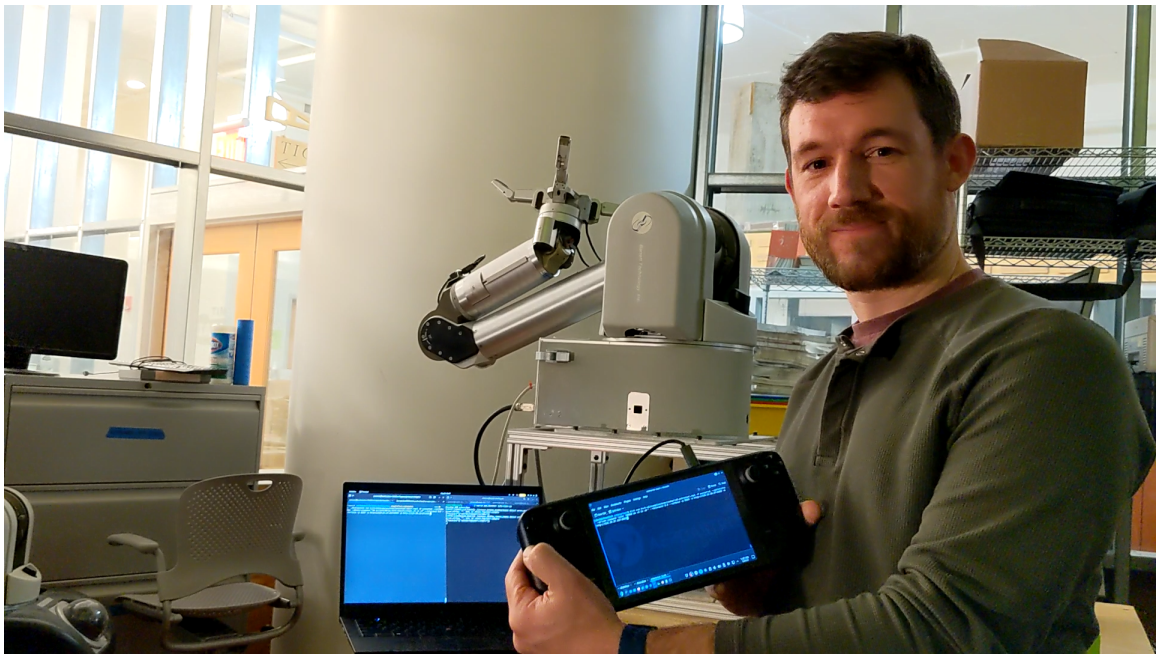


Figure 7-3: The two Kirks and two agents of the hardware demonstration. The Kirk executive running on the laptop is controlling the Barrett WAM arm in the background. Cameron Pittman is the second agent (acting as the astronaut) and interacting with a Kirk executive running on the Steam Deck handheld PC. This image was taken in the MERS lab on 20 May 2023.

The architecture of the hardware demonstration is as shown in Figure 7-4. We ran two Kirks on the same network. One Kirk was responsible for driving the Barrett WAM, while the other acted as the decision making logic behind an interface on the astronaut's person (say a tablet, heads-up-display, or portable computer of some kind).

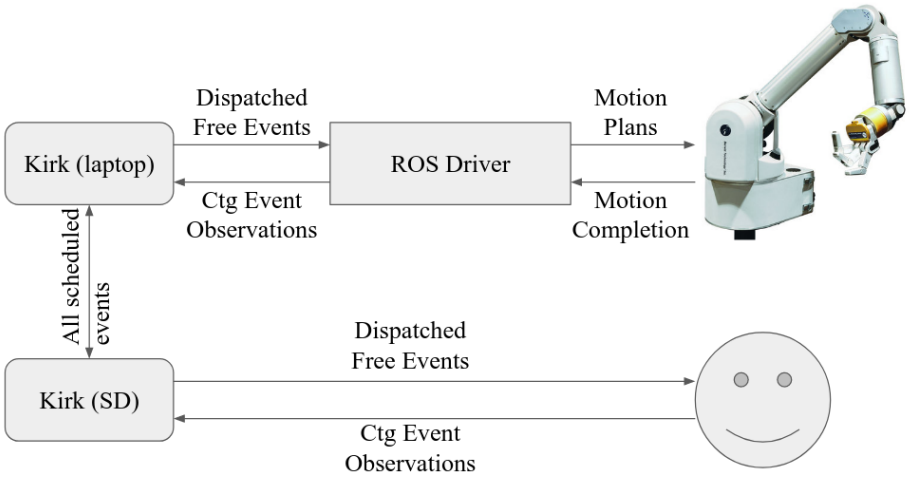


Figure 7-4: The information flow between the two agents and two Kirks in the hardware demonstration. "SD" is short for Steam Deck.

The laptop ran the Kirk that controls the WAM. It did so by dispatching requirement events to a separate driver that could translate event names to pre-built trajectories for the WAM. The trajectories were then published to the WAM's controller as ROS messages. As trajectories were completed, ROS messages were received by the ROS driver layer, which then sent contingent event observations back to Kirk.

The Valve Steam Deck (SD), a handheld PC, ran the astronaut's Kirk. The Kirk command line tool allows users to press a number to trigger the observation of a contingent event. We modified the output of the video game controller buttons of the Steam Deck such that they would automatically input the number corresponding to the observation of a contingent event. As the astronaut, Cameron only needed to press one button (mapped to left on the d-pad) during the run to trigger the observation.

The laptop and the Steam Deck were on the same local network. Note the cable dangling from the Steam Deck in Figure 7-3, which is a USB-C to Ethernet adapter.

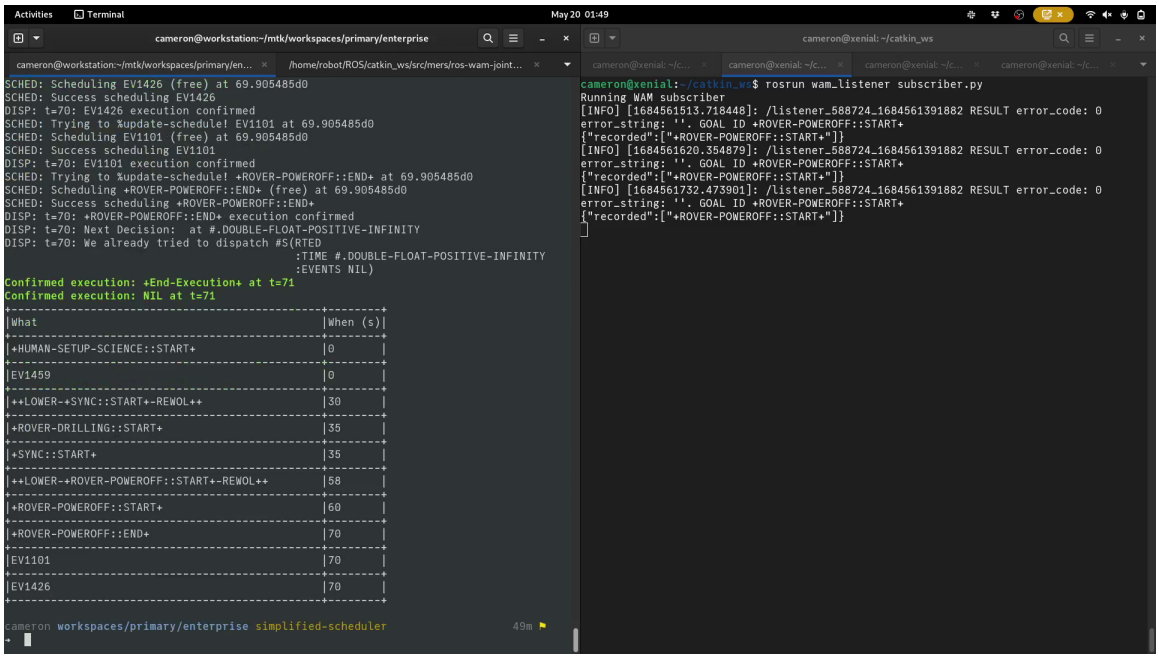


Figure 7-5: The laptop screen at the end of the second hardware demo scenario. On the left is Kirk's output, on the right is the ROS translation layer. Kirk is showing the schedule that it executed, while we can see logs from messages sent between Kirk and the ROS layer on the right.

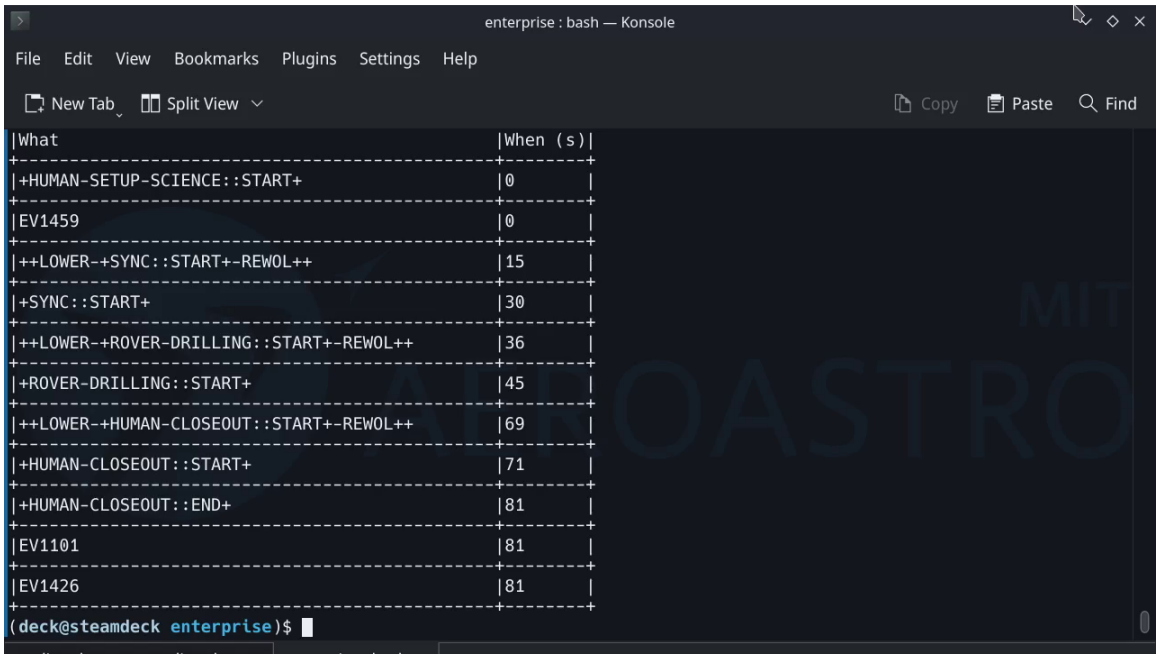


Figure 7-6: The Steam Deck screen at the end of the second hardware demo scenario. On the left is Kirk's output, on the right is the ROS translation layer. Kirk is showing the schedule that it executed, while we can see logs from messages sent between Kirk and the ROS layer on the right.

The Steam Deck was hardwired to the network for demonstration purposes. Communications occurred over HTTP. To simulate uncertain communication, a sleep call with a time in the range of $\bar{\gamma}(x_c)$ was injected into Kirk's function responsible for broadcasting event observations to peers, where $\bar{\gamma}(x_c)$ was drawn from any contingent event $x_c \in X_c$ of the receiving agent.

To start a run of the demonstration, we would start both Kirks simultaneously. Each Kirk had their own RMPL control program, which we include in Listings 5 and 6. Note that the control programs are nearly identical. The control programs related to the high bandwidth handoff, `human-downlink-science`, `sync`, and `robot-drilling`, differ only in observation delay and whether the `sync` event is controllable. Adding observation delay reflects uncertain communication between the agents.

The `sync` control programs were included as synchronization episodes between `human-downlink-science` and `robot-drilling`. Note that the robot also has a `sync` episode, which ensures that both agents agree on the naming of events.

We can see the modeling power of variable observation delay in Listings 5 and 6. It is natural that the observation delay between agents may change due to the evolution of resources during a mission. The variable-delay modeling framework allows us to model uncertain delay for each temporal constraint independently. For instance, if we know that, say, agents will be distant during a given constraint, then we may add uncertain delay accordingly. If agents are collocated during other constraints, then we can safely decrease the observation delay (absent other sources of delay).

According to the constraints and variable delay of the `human-downlink-science` control program from the perspective of the robot, the transformed fixed-delay STNU the robot is executing will reflect constraints of `human-downlink-science:start` $\xRightarrow{[30,35]}$ `human-downlink-science:end` with $\gamma($`human-downlink-science:end`$) = 0$ after applying Lemma 10.

We performed two demonstrations. In the first, the astronaut would observe the end of the science downlink, the end event of `human-downlink-science:end`, which would trigger a delayed observation being passed to the robot. Once the robot received the command, it would begin its `robot-drilling` activity. It passed observations of

```
(defpackage #:scenario1)

(in-package #:scenario1)

(define-control-program human-downlink-science ()
  (declare (primitive)
           (duration (simple :lower-bound 15 :upper-bound 30)
                     :contingent t)))

(define-control-program sync ()
  (declare (primitive)
           (duration (simple :lower-bound 5 :upper-bound 15
                             :min-observation-delay 0
                             :max-observation-delay 1)
                     :contingent t)))

(define-control-program robot-drilling ()
  (declare (primitive)
           (duration (simple :lower-bound 22 :upper-bound 26
                             :min-observation-delay 0
                             :max-observation-delay 2)
                     :contingent t)))

(define-control-program human-closeout ()
  (declare (primitive)
           (duration (simple :lower-bound 10 :upper-bound 30))))

(define-control-program main ()
  (with-temporal-constraint (simple-temporal :upper-bound 480)
    (sequence (:slack nil)
      (human-downlink-science)
      (sync)
      (robot-drilling)
      (human-closeout))))
```

Listing 5: The control program the astronaut uses while collecting and downlinking scientific data.

```
(defpackage #:scenario1)

(in-package #:scenario1)

(define-control-program human-downlink-science ()
  (declare (primitive)
           (duration (simple :lower-bound 15 :upper-bound 30
                             :min-observation-delay 5
                             :max-observation-delay 15)
                     :contingent t)))

(define-control-program sync ()
  (declare (primitive)
           (duration (simple :lower-bound 5 :upper-bound 15))))

(define-control-program robot-drilling ()
  (declare (primitive)
           (duration (simple :lower-bound 22 :upper-bound 26
                             :min-observation-delay 0
                             :max-observation-delay 1)
                     :contingent t)))

(define-control-program robot-poweroff ()
  (declare (primitive)
           (duration (simple :lower-bound 10 :upper-bound 30))))

(define-control-program main ()
  (with-temporal-constraint (simple-temporal :upper-bound 480)
    (sequence (:slack nil)
      (human-downlink-science)
      (sync)
      (robot-drilling)
      (robot-poweroff))))
```

Listing 6: The control program the robot uses to decide when to act with respect to learning the astronaut has finished collecting scientific data.

its scheduled events back to the astronaut.

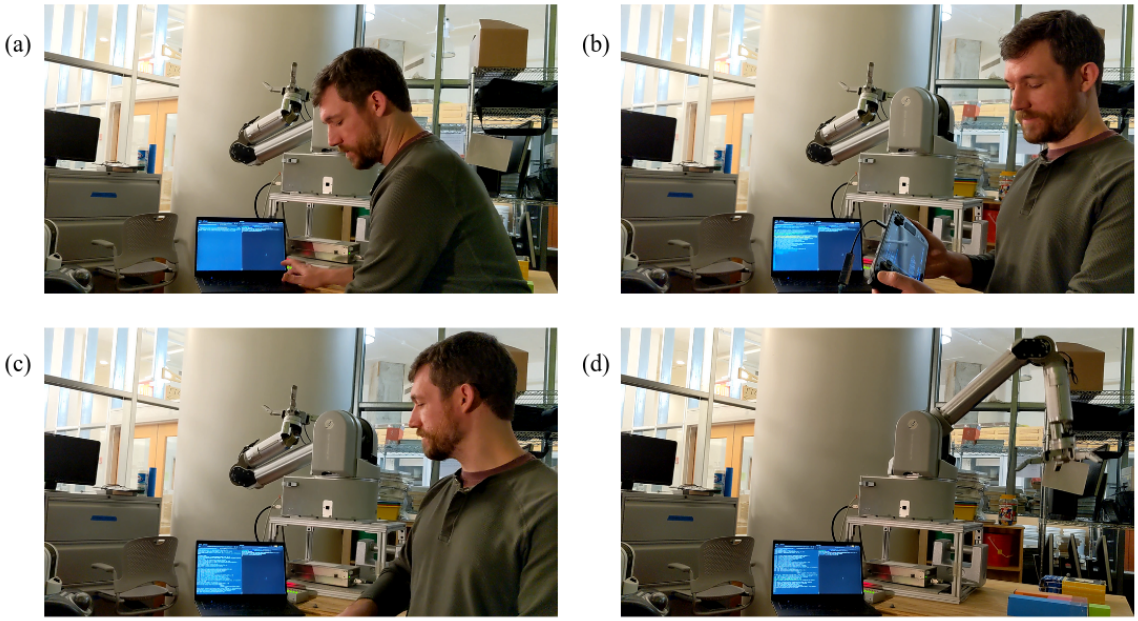


Figure 7-7: The first demonstration in four parts. (a) $t = 0$, when the two Kirks are started at the same time (unfortunately, the SD is below the image frame). (b) $t = 16$, when the astronaut observed that the science experiment was setup. (c) $t = 23$, when the robot received a delayed observation from the astronaut indicating they had completed science setup. (d) $t > 23$, as the robot performed the drilling task.

The second demonstration focused on the behavior of the delay scheduler when communications are not received in time. After starting both Kirks at the same time, we unplugged the astronaut's Kirk from the network. While, in reality, a disconnection should be modeled as $\bar{\gamma}^{+}(x_c) = \infty$, we did it to emphasize the fact that the robot's Kirk would not receive an observation within the fixed bounds it was expecting. The delay scheduler would then imagine `human-downlink-science:end` and dispatch its drilling activity accordingly.

We present the schedules of the agents for both scenarios in Tables 7.5-7.8. The schedule has been cleaned and the event names have been modified to better reflect the intent of the RMPL control programs. See Appendix B for an explanation of how RMPL and the STNUs compiled from it are related. We also removed anonymous (non-named) events that were added in the process of translating RMPL to variable-

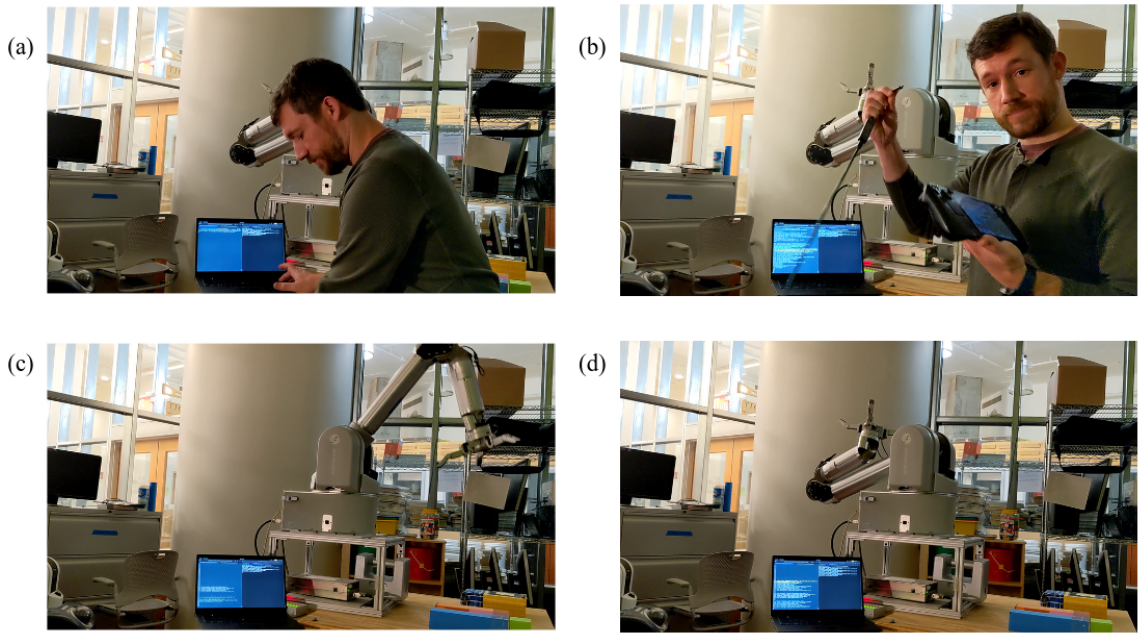


Figure 7-8: The second demonstration in four parts. (a) $t = 0$, when the two Kirks are started at the same time (unfortunately, the SD is below the image frame again). (b) $t = 3$, when the SD is removed from the network. (c) $t = 38$, after the robot imagined an observation from the astronaut and began the drilling task. (d) $t = 60$, when Kirk has observed the end of the drilling task.

delay STNU.

Table 7.5: The complete history of the delay scheduler for the astronaut in the first hardware demo scenario.

Event	**Time (s)**
Start Human Setup Science	0
Normalized Lower for End Human Setup Science	15
End Human Setup Science	16
Start Sync	16
Normalized Lower for Sync	22
End Sync	23
Start Robot Drilling	23
Normalized Lower for Robot Drilling	47
End Robot Drilling	49
Start Close Out	49
End Close Out	59

Table 7.6: The complete history of the delay scheduler for the robot in the first hardware demo scenario.

Event	**Time (s)**
Start Human Setup Science	0
Normalized Lower for Human Setup Science	15
End Human Setup Science	22
Start Sync	22
End Sync	22
Start Robot Drilling	22
Normalized Lower for Robot Drilling	45
End Robot Drilling	46
Start Robot Poweroff	46
End Robot Poweroff	56

We can see variable observation delay at work by comparing the assigned end times for `human-setup-science` between the astronaut and robot in Tables 7.5 and 7.6. The human knows that science setup was completed at $t = 16$, but the robot received an observation of the same event after an apparently delay of six seconds.

Note that the robot was running an experimental *optimistic* version of the delay scheduler in the demonstration. The difference between the delay scheduler as described in Chapter 5 and the optimistic version is that the optimistic version will attempt to avoid buffering early contingent events by rewriting the delay STNU based

on the early observation and checking VDC. It will be discussed in more depth in Appendix C.

Table 7.7: The complete history of the delay scheduler for the astronaut in the second hardware demo scenario.

Event	**Time (s)**
Start Human Setup Science	0
Normalized Lower for End Human Setup Science	15
End Human Setup Science	30
Start Sync	30
Normalized Lower for Sync	36
End Sync	45
Start Robot Drilling (imagined)	45
Normalized Lower for Robot Drilling	69
End Robot Drilling (imagined)	71
Start Close Out	71
End Close Out	81

Table 7.8: The complete history of the delay scheduler for the robot in the second hardware demo scenario.

Event	**Time (s)**
Start Human Setup Science	0
Normalized Lower for Human Setup Science	30
End Human Setup Science (imagined)	35
Start Sync	35
End Sync	35
Start Robot Drilling	35
Normalized Lower for Robot Drilling	58
End Robot Drilling	60
Start Robot Poweroff	60
End Robot Poweroff	70

Finally, we see in the second experiment that the robot was forced to imagine contingent events due to communication delay. The robot was able to satisfy all constraints with its drilling episode despite not receiving communication from the astronaut about the end of the astronaut setting up the science experiment.

Chapter 8

Future Work and Conclusion

The execution decisions produced by the delay scheduler are dependent on the strategy for checking VDC as presented in Chapter 4. While the delay scheduler has been shown to be effective, there is room for improvement. The delay scheduler presented in this thesis does not use the entirety of the execution space available to it. To demonstrate how execution space is limited, consider the following example.

$$A \xrightarrow{[1,3]} B \xrightarrow{[5,9]} C$$

Let $\bar{\gamma}(B) = [3, 8]$. After applying Lemmas 10 and 13, its fixed-delay equivalent representation is,

$$A' \xrightarrow{[6,9]} B' \xrightarrow{[4,6]} C'$$

where $\gamma(B) = 0$. According to the semantics of the original variable-delay STNU, B may arrive as early as $t = 4$. If so, the delay scheduler will compare the observation to the fixed-delay equivalent representation, forcing it to buffer B' to $t = 6$. C' will be scheduled no earlier than 4 time units later (the lower bound of $B' \xrightarrow{[4,6]} C'$) at $t = 10$.

However, if we were able to schedule the original variable-delay STNU directly, we could come to a different scheduling decision. If B arrives at $t = 4$, we know it must be the case that B was assigned at $t = 1$ and $\bar{\gamma}(B)$ resolved to a delay of 3. We

can then schedule C 5 time units after 1 (the lower bound of $B \xrightarrow{[5,9]} C$) at $t = 6$.

Clearly then, there is room for improvement in our execution strategy. I can imagine two avenues for future work. The first would be identifying a dispatchable form for variable-delay STNUs that fully encompasses their entire execution semantics. A dispatchable form that includes all resolutions of observation uncertainty would potentially allow us to schedule variable-delay STNUs directly, as opposed to the dance we do by observing resolutions of uncertainty from the full variable-delay STNU, but scheduling the limited fixed-delay equivalent. The second avenue would be to perform what we have be framing as *Optimistic Rescheduling*. We began work on Optimistic Rescheduling for this thesis. We have a partial implementation of it in Delay Kirk, though there are some caveats. First, it is largely untested and we have not shown it is sound and complete, hence why it was not included in this thesis. Second, it is seemingly inefficient and may inhibit a scheduler from performing its duties in larger STNUs.

We present our work on Optimistic Rescheduling so far in Appendix C. At a high level, the algorithm works roughly as follows.

1. Given an early observation of an uncontrollable event, x_c, at time t, rewrite the variable-delay STNU to narrow the temporal constraint and observation delay of x_c to the range that would allow an observation at time t.
2. Check VDC of the rewritten variable-delay STNU. If uncontrollable, abort.
3. Build a dispatchable form from the fixed-delay equivalent STNU.
4. Loop through all observed events earlier than t and perform the same observations in the same order against the new dispatchable form.
5. Observe x_c at time t in the new dispatchable form.

As shown in Appendix C, the total time to schedule all events is $O(\frac{1}{2}N^2(N + 1)\log N)$ in the worst case due to step 4.

Ideally, we would not need to reschedule past events to change future constraints. Future work could entail finding a procedure for tightening the current dispatchable form directly instead of rescheduling the entire run from scratch.

What if we looked for conflicts? Could we possibly search for a time to dispatch early in the buffered execution space?

Smarter rewriting STNU. Could we update the existing d-graph directly and check it for SRNCs? maybe?

8.1 Coordination

We were focused on addressing the multi-agent (MA) online scheduling problem. Before scheduling, we must contend with planning, e.g. building variable-delay STNUs for each agent. We considered extending the two existing planning approaches described below to model variable observation delay between agents. We ultimately decided neither were fit for the motivating scenarios of this thesis. Instead, we used a manual planning approach more akin to the ISS EVA planning process.

The first planning approach we considered was to model the system as a Multi-Agent STNU (MASTNU) [11]. MASTNUs allow modelers to describe temporal constraints between multiple agents, then check the overall dynamic controllability of the system. To check the controllability of a MASTNU, the first step is to perform temporal decoupling with the goal of producing individual dynamically controllable STNUs for each agent that can be dynamically scheduled per usual. While superficially promising, there is a considerable drawback to this approach, namely that temporal decoupling is sound but not complete, i.e. temporal decoupling may report failure even when the MASTNU is dynamically controllable. This limits the utility of MASTNUs as a planning tool.

The other approach to this problem we are aware of is Stedl's Hierarchical Reformulation (HR) algorithm [44]. HR begins with a MA temporal plan network (TPN), which is similar to a MASTNU (though HR pre-dates MASTNUs). Stedl's key insight is to avoid inter-agent communication altogether by reformulating constraints between groups of agents such that they are strongly controllable. As such, no communication between agents is required. A centralized dispatcher is then responsible for then handing events to agents. We also assume that there is no central authority,

making HR a poor fit for our problem domain.

Both MASTNUs and HR assume communications between agents are either instantaneous or impossible, i.e. with an infinite delay. As we will see in Section 4.3, our formalism for variable observation delay allows a *spectrum* of communication delay. While we felt it was possible to shoehorn uncertain observation delay into MASTNUs or HR, we felt both were a poor choice because of their pre-existing expectations with respect to communication. In combination with our focus on online scheduling, we decided to forgo extending either formalism to account for observation delay. Instead our planning process simply consists of manually writing variable-delay STNUs with intra-agent and inter-agent temporal constraints by hand.

We believe it may be possible for MASTNUs or HR to be expanded to include variable observation delay, though we leave that problem for future research.

We considered framing our approach to inter-agent communication as a distributed consensus problem because we believed we needed a means for disparate agents to agree on the state of the world. Existing distributed consensus algorithms like Paxos [43] or Raft [42] would then be integrated into the communication layer of Kirk and take responsibility for ensuring that agents agree on which events have been scheduled.

Ultimately the drawbacks of a distributed consensus approach outweighed the benefits. Chiefly, both Paxos and Raft assume that communications are either instantaneous and freely available or that agents have gone dark (i.e. can no longer communicate). This communication model is incongruous with the explicitly modeled communications of the VDC formalism. Furthermore, the VDC formalism allows us to model that agents never receive communications, negating the requirement for distributed consensus.

8.2 Conclusion

Appendix A

Comparison of Variable-Delay STNUs to Partially Observable STNUs

The delay scheduler is flexible in that so long as it receives a variable-delay STNU, it is capable of scheduling. Human modelers have flexibility in how they represent temporal constraints in that there are many flavors of STNUs, each with their own advantages and disadvantages. Earlier, we presented RMPL as a modeling language that is compiled to variable-delay STNUs. There are other choices for modeling frameworks. Here, we present a comparison of variable-delay STNUs to POSTNUs [12], a flavor that is similar in many respects. This section presents a comparison of variable-delay STNUs and POSTNUs, including transformations that allow some classes of POSTNUs to be represented as variable-delay STNUs.

One of the strengths of the variable-delay controllability model is its ability to generalize the concepts of strong and dynamic controllability. This technique was first seen in greater depth in the context of POSTNUs. In an STNU, all contingent events are either instantaneously observable under a dynamic controllability model or entirely unobservable under a strong controllability model. In POSTNUs, contingent events can be marked observable and unobservable. To say that a POSTNU is dynamically controllable equates to asserting that it is possible to construct a schedule

during execution that respects all constraints if the scheduler only receives information about observable contingent events.

While, superficially, POSTNUs and delay STNUs appear to model distinct problems in temporal reasoning, all delay STNUs can be accurately represented as POSTNUs. While the converse is not true, there is a subset of POSTNUs that delay STNUs are able to model. It is advantageous to translate POSTNUs to delay STNUs when possible because we are guaranteed to finish controllability checks for delay STNUs in polynomial time, while evaluating the controllability of POSTNUs in general is harder [12]. Furthermore, the sub-class of POSTNUs that can be checked efficiently and accurately, those without chained contingent links [45], are members of the subset of POSTNUs that can be expressed directly as STNUs with fixed-delay, and likewise variable-delay, functions , [29, p. 59]. The converse is also true - we may emulate STNUs with variable-delay functions as POSTNUs without chained contingent links. Below, we elaborate on translations from delay STNUs to POSTNUs, before describing how we can express POSTNUs without chained contingent links as STNUs with fixed-delay functions. Note that this is not a comprehensive list of transformations between delay STNUs and POSTNUs - our aim is to describe the minimum set of transformations required to model POSTNUs without chained contingent links as delay STNUs. For the following discussion, let S be a delay STNU and P be an equivalent POSTNU.

We present these transformations to demonstrate that the delay scheduler is capable of scheduling POSTNUs without chained contingencies. So long as it receives a variable-delay STNU, the fact that POSTNUs can be scheduled allows modelers additional flexibility in their means of representing the problem domain.

We start with an S that consists of the links $A \Rightarrow B$ and $B \rightarrow D$ with delay function $\bar{\gamma}(B)$. See Figure A for an example translation between a fixed-delay STNU and a POSTNU.

Lemma 18. *For contingent link $A \Rightarrow B$ in S with observation delay $\bar{\gamma}(B) = [l, u]$, where $0 \leq l \leq u < \infty$, and outgoing requirement link $B \rightarrow D$, we may emulate observation delay in P by copying $A \Rightarrow B$ and $B \rightarrow D$ to P, enforcing that B is*

unobservable, and adding a new observable contingent event, B' with $B \overset{[l,u]}{\Longrightarrow} B'$.

Proof. In both S and P, we do not observe B directly, yet we define an outgoing requirement link, $B \to D$, that depends entirely on the resolution of B. The only information available to reason about the assignment of B comes in the form of an indirect observation, B' or $\bar{\gamma}(B)$, received after a delay in $\mathbb{R}^{\geq 0}$. If P was not equivalent to S, we would be able to learn the assignment of B without waiting $\bar{\gamma}(B)$ time units after its true assignment. Thus, because we must wait $B' - B \in [l, u]$ time units to learn the assignment of B, and $B \to D$ has equivalent constraints between S and P, P must model the same semantics as S. □

Lemma 19. *For a contingent event B with variable-delay function $\bar{\gamma}(B) = [0, 0]$ in S, we may emulate the same constraints with an observable contingent event, B in P.*

Proof. The variable-delay function enforces instantaneous observation. By the definition of observable contingent events in the POSTNU model, we will observe B instantaneously. □

A POSTNU with a chained contingency is defined as follows. Consider a chain of contingent events, $A \Rightarrow B$ and $B \Rightarrow C$. If B has one or more outgoing links to free or contingent events other than C, it is a chained contingency. Lemma 18 results in a POSTNU without chained contingencies, hence variable-delay STNUs fall into the subclass of POSTNUs that can be checked efficiently [45].

In the other direction, to transform a POSTNU without chained contingencies into an STNU with variable-delay functions, we need to address three cases of contingent constraints: (1) unobservable contingent events not immediately followed by other contingent constraints, (2) unobservable contingent events immediately followed by other contingent constraints, and (3) observable contingent constraints.

Lemma 20. *For an unobservable event B in P, with no outgoing contingent links, we can emulate it in S with a contingent event B and upper bound of its variable-delay function set to $\bar{\gamma}^+(B) = \infty$.*

Proof. We will not observe B nor will outgoing contingent links provide information about B. As such we define $\bar{\gamma}^+(B) = \infty$ in S.[1] From a controllability standpoint for both S and P, we know the *a priori* bounds of B but will not learn its true assignment. □

Lemma 21. *For an unobservable contingent link $A \xRightarrow{[m,n]} B$ in P, with a single outgoing link, $B \xRightarrow{[w,z]} C$, we replace the two constraints from P in S with a concatenated constraint, $A \xRightarrow{[m+w,n+z]} C$.*

Proof. The only information we may receive is the observation of C. Given there are no other outgoing links from B, folding B into the successive contingent constraint can not affect the semantics of the network. The bounds of the new link, $A \xRightarrow{[m+w,n+z]} C$ is the result of summing the intervals of $A \xRightarrow{[m,n]} B$ and $B \xRightarrow{[w,z]} C$: $[m, n] + [w, z] = [m + w, n + z]$. □

Note that we did not specify whether C is observable in P. After applying Lemma 21, we then apply either Lemma 19 or 20 to C.

Lemma 22. *For an observable contingent event $A \xRightarrow{[m,n]} B$ in P, with a single outgoing link to an observable contingent event, C, $B \xRightarrow{[w,z]} C$, we create three constraints in S: $A \xRightarrow{[m,n]} B$, $B \xrightarrow{[0,0]} B'$, and $B' \xRightarrow{[w,z]} C$ where $\bar{\gamma}(B) = [0, 0]$.*

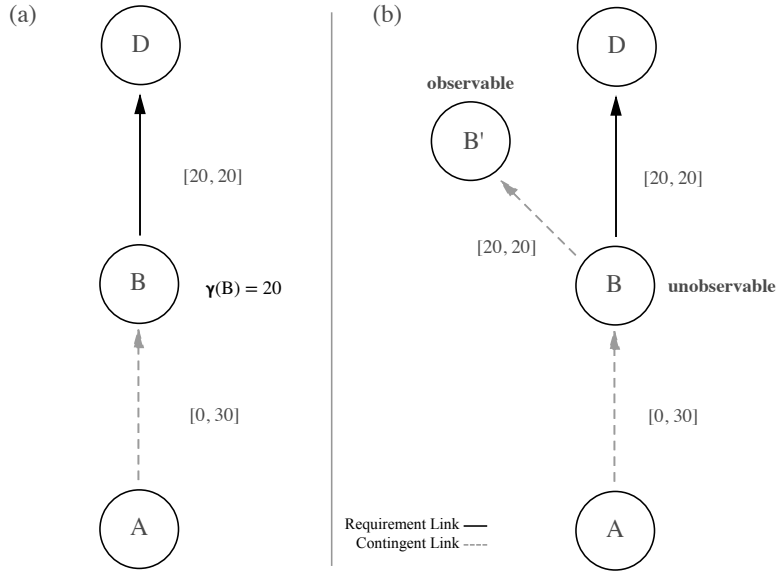
Proof. Given C is observable in P, a simulated free event, B' in S, can be scheduled simultaneously with B. Any contingent constraints following B now start at an executable event and are thus valid constraints in S. B is observable, so we need no observation delay according to Lemma 19. □

Thus, delay STNUs are sufficiently capable of expressing all POSTNUs that can efficiently be checked for controllability using today's tractable POSTNU algorithms.

It is not clear if controllability can be checked more efficiently across a greater subset of POSTNUs beyond those without chained contingencies. However, it is worth highlighting that variable-delay controllability can be leveraged to construct improved

[1] Note: ,p. [29, p. 60] erroneously claims that we should define $\gamma(B) = 0$ for unobservable events.

Figure A-1: (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.



algorithms with respect to scheduling and controllability of POSTNUs. The model for observation delay proposed by variable-delay controllability can be expressed exactly as a POSTNU with a "single-headed" chained contingency[2] as shown in Figure Ab; the main difference is that we represent the contingent link between B and B' with our variable-delay function $\bar{\gamma}(B)$. Hence, the algorithm we present for variable-delay controllability can be used to both solve POSTNUs without chained contingencies, as described above, as well as those POSTNUs with single-headed chained contingencies. Approaches inspired by variable-delay controllability have been used to further expand POSTNU dynamic controllability checking in more expressive chained instances [46]. We hope that insights from variable-delay controllability will continue to expand the subset of POSTNUs that can be controllability checked, and as such we advocate for continued development of the theory of variable-delay controllability as a relevant framework for modelers.

[2]To borrow the term "single-headed" from [46].

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix B

Additional RMPL Information

Modeling constraint programs in RMPL was required for the multi-agent demonstrations of a delay scheduler. We include additional information on RMPL below.

B.1 Control Programs and STNUs

Key to the challenge of being a mission planner is to understand the constraints between events. RMPL is an abstraction layer, meaning that it will interpret RMPL and produce a delay STNU reflecting its own interpretation, which may not align with the mission planner's intention. We have seen students in the lab be surprised by the temporal constraints generated by RMPL, and, through the experiments involved in this thesis, have also found RMPL to be less-than-intuitive on occasion. The chief cause of confusion is a disjoint between the fundamental unit of temporal reasoning, the temporal event, and the fundamental unit of RMPL control programs, the episode, which consists of a start and an end event. This is not a limitation of the semantics of RMPL, rather, it is a naming issue.

To elaborate, temporal reasoning literature emphasizes events as meaningful timepoints. For instance, we used numbered `Confirm:0:0` events to represent the end of an installation task in Sections 5.3 and 7.3.1, while the start of the task was an `Install:0:0` event, e.g. `Install:0:0` $\xrightarrow{[l,u]}$ `Confirm:0:0`. Depending on the size of the generated STNU, the `Confirm:0:` event then had outgoing edges to events such

as `Start:0:1`, `Install:1:0`, or `ALL:END`. In RMPL, however, a representation of this constraint between events with these names is impossible.

Consider the following control program.

```
(define-control-program take-pictures ()
  (declare (primitive)
           (duration (simple :lower-bound 1 :upper-bound 5))))
```

We have defined an episode named `take-pictures`. It represents a requirement constraint in the form of `take-pictures:start` $\xrightarrow{[1,5]}$ `take-pictures:end`. Note that the one episode led to the creation of two events. We now add a second episode and show the two options RMPL gives us for joining them in series.

```
(define-control-program eat-snack ()
  (declare (primitive)
           (duration (simple :lower-bound 3 :upper-bound 7))))

;; Sequence option 1
(define-control-program with-slack ()
  (with-temporal-constraint (simple-temporal :lower-bound 6 :upper-bound 10)
    (sequence (:slack t)
              (take-pictures)
              (eat-snack))))

;; Sequence option 2
(define-control-program no-slack ()
  (with-temporal-constraint (simple-temporal :lower-bound 6 :upper-bound 10)
    (sequence (:slack nil)
              (take-pictures)
              (eat-snack))))
```

We have two commonly used options for `:slack` in the sequence, `(:slack nil)` and `(:slack t)`. If slack is applied, episodes are connected with $[0, \infty]$ constraints,

effectively guaranteeing episode ordering without enforcing that the latter episode must start immediately after the first. Ignoring the overall constraint, the two control programs are connected like so.

$$\texttt{take-pictures:start} \xrightarrow{[3,7]} \texttt{take-pictures:end} \xrightarrow{[0,\infty]} \texttt{eat-snack:start} \xrightarrow{[3,7]} \texttt{eat-snack:end}$$

Without slack, the constraints are arranged as follows.

$$\texttt{take-pictures:start} \xrightarrow{[3,7]} \texttt{take-pictures:end} \xrightarrow{[0,0]} \texttt{eat-snack:start} \xrightarrow{[3,7]} \texttt{eat-snack:end}$$

The RMPL compiler takes it a step further. A constraint of the form $A \xrightarrow{[0,0]} B$ enforces simultaneous execution of A and B, making one of the events redundant from a scheduling perspective. In this example, RMPL removes `take-pictures:end`, leaving us with three events for the two constraints.

$$\texttt{take-pictures:start} \xrightarrow{[3,7]} \texttt{eat-snack:start} \xrightarrow{[3,7]} \texttt{eat-snack:end}$$

From an execution perspective, `eat-snack:start` both signifies the end of taking pictures and the beginning of snacking.

Going back to the installation example, we want the execution semantics of deciding when to start, traversing to an installation location, performing the installation, waiting for the response, and then moving to the next installation. As described in Section 5.3, the delay STNU takes the form as follows, with $\bar{\gamma}(\texttt{confirm:0:0}) = [0, 3]$.

$$\texttt{start:0:0} \xRightarrow{[1,15]} \texttt{traverse:0:0} \xrightarrow{[1,14]} \texttt{install:0:0} \xRightarrow{[1,6]} \texttt{confirm:0:0} \xrightarrow{[2,12]} \texttt{start:0:1}$$

In RMPL, we model each iteration of the procedure like so. (We use the `/0-0` numbering instead of the `:0:0` scheme in RMPL because `:` causes errors related to

lisp package exports.)

```
(define-control-program traverse/0-0 ()
  (declare (primitive)
           (duration (simple :lower-bound 1 :upper-bound 15)
                     :contingent t)))

(define-control-program install/0-0 ()
  (declare (primitive)
           (duration (simple :lower-bound 1 :upper-bound 14))))

(define-control-program confirm/0-0 ()
  (declare (primitive)
           (duration (simple :lower-bound 1 :upper-bound 6
                             :min-observation-delay 0 :max-observation-delay 3)
                     :contingent t)))

(define-control-program start/0-1 ()
  (declare (primitive)
           (duration (simple :lower-bound 2 :upper-bound 12))))

(define-control-program main ()
  (with-temporal-constraint (simple-temporal :upper-bound 108)
    (sequence (:slack nil)
      (traverse/0-0)
      (install/0-0)
      (confirm/0-0)
      (start/0-1))))
```

This results in a delay STNU of the form below (ignoring the overall upper-bound for simplicity's sake).

$$\texttt{traverse/0-0:start} \xRightarrow{[1,15]} \texttt{install/0-0:start} \xrightarrow{[1,14]} \texttt{confirm/0-0:start} \xRightarrow{[1,6]}$$
$$\texttt{start/0-1:start} \xrightarrow{[2,12]} \texttt{start/0-1:end}$$

If we need to translate events between the two forms, we simply note that event names are shifted by one place earlier in the delay STNU generated by RMPL.

B.2 Action Model

This section is included to expand on the features of RMPL, though note that none of these features are required for controlling distributed agents, and were not a part of the experiments for this research.

If we wanted to specify agents in a multi-agent control program, or if we wanted to take vehicle dynamics into account, RMPL gives us a means for using the Common Lisp Object System (CLOS) for defining agents, agent dynamics, and the control programs agents may execute.

An example RMPL control program with an agent is provided in Listing 7 for completeness sake from the domain of underwater robotics.

In Listing 7, `glider` refers to a low-powered autonomous underwater vehicle that prefers to traverse by following ocean currents using a buoyancy engine.[1] We see that we model a `glider` agent and its properties using standard CLOS. The `move` control program then takes a `glider` and a `location` as arguments. The `(requires ...)` form is equivalent to the preconditions of a durative action in a PDDL 2.1 [47] domain. Likewise, the `(effect ...)` form is equivalent to PDDL effects. Finally, as we saw before, the durative action also includes a temporal constraint in its `(duration ...)` form.

Kirk is able to take RMPL as input to perform classical planning, though further discussion of it falls outside the scope of this thesis.

[1]The Slocum Glider is an example: https://www.whoi.edu/what-we-do/explore/underwater-vehicles/auvs/slocum-glider/.

```
;; This code is a snippet from a file in the thesis code repo found at:
;;      kirk-v2/examples/glider/script.rmpl

(defclass glider ()
  ((id
    :initarg :id
    :finalp t
    :type integer
    :reader id
    :documentation
    "The ID of this glider.")
   (deployed-p
    :initform nil
    :type boolean
    :accessor deployed-p
    :documentaiton
    "A boolean stating if the glider is deployed at any point in time.")
   (destination
    :initform nil
    :type (member nil "start" "end" "science-1" "science-2")
    :accessor destination
    :documentation
    "The location to which the glider is currently heading, or NIL if it is not
    in transit.")
   (location
    :initarg :location
    :initform "start"
    :type (member nil "start" "end" "science-1" "science-2")
    :accessor location
    :documentation
    "The location where the glider is currently located, or NIL if it is not at
    a location (in transit)."))))

(define-control-program move (glider to)
  (declare (primitive)
           (requires (and
                      (over :all (= (destination glider) to))))
           (effect (and
                    (at :start (= (destination glider) to))
                    (at :start (= (location glider) nil))
                    (at :end (= (destination glider) nil))
                    (at :end (= (location glider) to))))
           (duration (simple :lower-bound 10 :upper-bound 20))))
```

Listing 7: A snippet of an RMPL script that defines an agent and classical planning predicates and effects of a control program.

Appendix C

Optimistic Rescheduling

We return to problem of potentially unnecessary wait time created by the buffering execution strategy described in Lemma 11. First, we use an example to demonstrate how buffering early contingent events results in a reduction of the execution space. Then we contribute a technique for managing event observations that circumvents the loss of execution space.

Consider the following variable-delay controllable STNU, which we will refer to as *Bufferable*.

$$A \xrightarrow{[1,7]} \overset{\bar{\gamma}\in[1,3]}{B} \xrightarrow{[5,9]} C$$

Following the semantics of the delay scheduler, we would first transform *Bufferable* to its fixed-delay equivalent, *Bufferable*$'$ by applying Lemma 10.

$$A' \xrightarrow{[4,8]} \overset{\gamma=0}{B'} \xrightarrow{[4,6]} C'$$

If we assume A is executed at $t = 0$, the only question is when to schedule C (or its fixed-delay equivalent, C'). According to the semantics of *Buffering*, if B is observed at $t = 2$, we know that B was assigned at $t = 1$. Thus, we only need to wait until $t = 6$ to schedule C. However, the delay scheduler would schedule according the constraints found in *Buffering*$'$, wherein $\xi(B') = 2$ falls earlier than the lower bound of $A' \xrightarrow{[4,8]} B'$, triggering Lemma 11. As a result, we act as if $\xi(B') = 4$ and then wait

for the lower bound of $B' \xrightarrow{[4,6]} C'$. The end result is that C' is assigned to a later time of $t = 8$.

From a human mission manager perspective, this wait appears to be a waste. Time is money. And in the case of planetary exploration, time is safety. If a NASA flight controller were to ask why your software is telling astronauts on Moon to just stand there doing nothing, responding that your algorithm *does not know* if it is safe to act, would be unacceptable. Therefore, we contribute a generate-and-test approach that looks for opportunities to avoid buffering when contingent events arrive before their expected windows in the fixed-delay STNU. The goal of this method is to dispatch future events earlier if possible.

At its core, optimistic rescheduling consists of copying the original variable-delay STNU then rewriting it to reflect the resolution of uncertainty so far. Key to rewriting the variable-delay STNU is narrowing the constraint and observation delay to match what was observed. We then re-perform controllability checks. If controllable, we have a new schedule that removes the need to buffer this contingent event. If not controllable, we do nothing, buffer the contingent event as planned, and continue dispatching against the original schedule.

We now step through the Event Observations with optimistic rescheduling algorithm (Algorithm 8) in detail.

We cannot know if an event is buffered if we do not attempt to schedule it. Our first step is to schedule an event like normal. If scheduling is possible without buffering, we simply return whether scheduling was successful.

If the event was buffered, then we begin to optimistically reschedule. We do so by tightening the bounds of the original VDC STNU, $S_{original}$, based on the observation we received, which is the responsibility of Algorithm 9, implementing Lemma 23.

If the rewritten STNU, S^*, is found to be VDC, we prepare to schedule it. First we iterate through all the assignments in the partial schedule and make the same assignments against the new STNU. When assignments are made, we subtract out the fixed observation delay. In this loop, we add the observation delay back, lest it be subtracted from the original observation twice.

Input: Original VDC STNU S; Equivalent fixed-delay function γ;
Partial history ξ; Executed events map $Ex(S, x)$; Observed contingent event x;
Normalized lower bound $\hat{x}$; Current time t;
Output: Boolean whether x was successfully scheduled, VDC STNU

Event Observations with Optimistic Rescheduling:

1 $successp, bufferedp \leftarrow$ `updateSchedule(S, x, t)`;
2 **if** $\neg bufferedp$ **then**
3 `return` $successp, S$;
4 **endif**
5 $S^* \leftarrow$ `rewriteSTNU(S, x, t)`;
6 **if** S^* *is not variable-delay controllable* **then**
7 `return` $successp, S$;
8 **endif**
9 **for** a *in* ξ `//`a `is an assignment`
10 **do**
11 **if** $\gamma(a[event]) \neq \infty$ **then**
12 `updateSchedule`$(S^*, a[event], a[time] + \gamma(a[event]))$;
13 **endif**
14 **end**
15 **for** *event in* $Ex(S)$ **do**
16 $Ex(S^*, x) \leftarrow Ex(S, x)$
17 **end**
18 `updateSchedule`$(S^*, \hat{x}, t)$;
19
20 `return true`, S^*;

Algorithm 8: An Algorithm for observing contingent events with optimistic rescheduling.

If any contingent events with infinite delay were observed, they would have been marked executed but not assigned. We iterate through the executed events of S and mark the same events executed in S^*.

The distance graph, partial schedule, and executed events of S^* now match that of S before x_c was received. We are almost safe to record a new observation. Lastly, we must address the executable event representing the normalized lower bound of x_c, $\hat{x}_c$. During scheduling, we would have received an RTED consisting of $\langle l+\bar{\gamma}^+(x_c), \hat{x}_c\rangle$. Given that x_c arrived before $l+\bar{\gamma}^+(x_c)$, we never would have assigned $\hat{x}_c$, so we assign $\xi(\hat{x}_c) = t$ now. We finally update the schedule with the contingent event that arrived.

Lemma 23. *If a contingent event, $x_c \in X_c$, where $u-l > \bar{\gamma}^+(x_c)-\bar{\gamma}^-(x_c)$, is observed at time t and when $t < l + \bar{\gamma}^+(x_c)$, we may replace x_c and $\bar{\gamma}(x_c)$ with a constraint, x_c^*, and variable-delay function, $\bar{\gamma}(x_c^*)$, with narrower bounds as follows.*

$$
\begin{aligned}
x_c^* &= [l^*, u^*] \\
x_c^* &= [\max(l, t - \bar{\gamma}^+(x_c)), \min(u, t - \bar{\gamma}^-(x_c))] \\
\bar{\gamma}(x_c^*) &= [\max(\bar{\gamma}^-(x_c), t - u), \min(\bar{\gamma}^+(x_c), t - l)]
\end{aligned}
$$

Proof. Buffering is only possible if the conditions of Lemmas 10 and 11 are triggered. By Lemma 10, we are guaranteed to be able to narrow where in the range $[l, u]$ x_c was scheduled. By Lemma 11, we know that rewritten bounds will lead to an assignment of x_c that is no later than $l + \bar{\gamma}^+(x_c)$. Our tool for narrowing the bounds is Equation 5.5, which allows us to use the observation to reason over the assignment and observation delay. Our strategy is to look at the extreme cases leading to an observation.

We start by reasoning over the earliest and latest assignments respectively. In order for x_c to be assigned as early as possible, l^*, we assume the delay has taken on its maximum value, $\bar{\gamma}^+(x_c)$.

$$\xi(x_c) = \texttt{obs}(x_c) - \gamma(x_c) \tag{C.1}$$

$$l^* = t - \bar{\gamma}^+(x_c) \tag{C.2}$$

Likewise, to find the last possible assignment leading to an observation, we subtract the smallest observation delay, $\bar{\gamma}^-(x_c)$.

$$u^* = t - \bar{\gamma}^-(x_c) \tag{C.3}$$

Given that Nature will adhere to the constraints originally put forth in S, the bounds of x_c^* must remain within the bounds of x_c. Hence, we guarantee the lower bound is at least l while the upper bound is at most u.

$$l^* = \max(l, t - \bar{\gamma}^+(x_c))$$
$$u^* = \min(u, t - \bar{\gamma}^-(x_c))$$

We use the same logic for narrowing the observation delay. If x_c was assigned as late as possible, u, then the observation delay would be minimized, $\bar{\gamma}^-(x_c^*)$. Likewise, if x_c was assigned as early as possible, l, the observation delay would be maximized, $\bar{\gamma}^+(x_c^*)$. The narrowed lower and upper bounds of $\bar{\gamma}(x_c)^*$ are as follows.

$$\gamma = \texttt{obs}(x_c) - \xi(x_c)$$
$$\bar{\gamma}^-(x_c^*) = t - u$$
$$\bar{\gamma}^+(x_c^*) = t - l$$

As before, the bounds of $\bar{\gamma}(x_c^*)$ must stay within the original bounds of $\bar{\gamma}(x_c)$,

leaving us with the following narrowed observation delay.

$$\bar{\gamma}^{-}(x_c^*) = \max(\bar{\gamma}^{-}(x_c), t-u) \tag{C.4}$$

$$\bar{\gamma}^{+}(x_c^*) = \min(\bar{\gamma}^{+}(x_c), t-l) \tag{C.5}$$

□

We revisit the example from the beginning of this section to see Lemma 23 in action. As we saw before, any $\texttt{obs}(B)$ before $t = 4$ will result in buffered assignments.

$$A \overset{[1,7]}{\Longrightarrow} {}^{\bar{\gamma}\in[1,3]}B \overset{[5,9]}{\longrightarrow} C$$

Let $t = 3$. We will step through the reasoning for narrowing the bounds of x_c accordingly.

$$\begin{aligned} x_c^* &\in [\max(l, t-\bar{\gamma}^{+}(x_c)), \min(u, t-\bar{\gamma}^{-}(x_c))] \\ x_c^* &\in [\max(1, 3-3), \min(7, 3-1)] \\ x_c^* &\in [1, 2] \end{aligned}$$

$$\begin{aligned} \bar{\gamma}(x_c^*) &\in [\max(\bar{\gamma}^{-}(x_c), t-u), \min(\bar{\gamma}^{+}(x_c), t-l)] \\ \bar{\gamma}(x_c^*) &\in [\max(1, 3-7), \min(3, 3-1)] \\ \bar{\gamma}(x_c^*) &\in [1, 2] \end{aligned}$$

We find that $\xi(x_c)$ must have fallen somewhere in the range of $[1, 2]$, while $\bar{\gamma}(x_c)$ was resolved somewhere in $[1, 2]$. Looking at the extremes, it is clear that there are multiple combinations of the assignment and observation delay that could lead to an observation at $t = 3$. While the narrowed range allows for observations other than $t = 3$, for instance, if $\xi(x_c) = 2$ and $\texttt{obs}(x_c) = 2$ yielding an observation at $t = 4$, there are no other ranges of assignments or observation delay outside of $\xi(x_c) \in [1, 2]$

and $\bar{\gamma}(x_c) \in [1, 2]$ that would allow an observation at $t = 3$.

Input: VDC STNU $S_{original}$; Variable-delay function $\bar{\gamma}$;
Observed contingent event x; Observation time t;
Output: VDC STNU
Initialization: $S_{new} \leftarrow$ `copy`$(S_{original})$

Rewrite STNU:

1 **for** *constraint in* S_{new} **do**
2 **if** *constraint ends in* x **then**
3 $constraint[lower] \leftarrow \max(constraint[lower], t - \bar{\gamma}^+(x))$;
4 $constraint[upper] \leftarrow \min(constraint[upper], t - \bar{\gamma}^-(x))$;
5 $\bar{\gamma}^-(x) \leftarrow \max(\bar{\gamma}^-(x), t - constraint[upper])$;
6 $\bar{\gamma}^+(x) \leftarrow \max(\bar{\gamma}^+(x), t - constraint[lower])$;
7 **endif**
8 **end**
9 `return` S_{new};

Algorithm 9: An Algorithm for rewriting an STNU given the resolution of uncertainty of a contingent link.

The complexity of Algorithm 8 is dominated by the loop over `updateSchedule`. Each call to `updateSchedule` is $O(N^3)$ in the number of events.

In the worst case, each of the N events could trigger Optimistic Rescheduling. We know that the total time to schedule one event scales with $O(N \log N)$. Scaling all events goes as $O(N^2 \log N)$. If we had to schedule the next to last event too, the performance would be $O(N^2 \log N + (N-1)N \log N)$. The last two events being rescheduled would have the performance of $O(NN \log N + (N-1)N \log N) + (N-2)N \log N)$, and so on for all N events. We know that $\sum_{n=0}^{N} N - n = \frac{1}{2}N(N+1)$, giving us $O(N^2(N+1) \log N)$ total runtime to schedule (and reschedule) all events.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] D. A. Coan, Exploration EVA System Concept of Operations, National Aeronautics and Space Administration, Houston, TX, EVA-EXP-0042 Revision B, 2020.

[2] J. W. McBarron, Past, present, and future: The U.S. EVA Program, *Acta astronautica*, vol. 32, no. 1, pp. 514, 1994, doi: 10.1016/0094-5765(94)90143-0.

[3] C. P. Sonnett, REPORT of the AD HOC WORKING GROUP ON APOLLO EXPERIMENTS AND TRAINING on the SCIENTIFIC ASPECTS OF THE APOLLO PROGRAM, NASA, 1963.

[4] J. M. Hurtado, K. Young, J. E. Bleacher, W. B. Garry, and J. W. Rice, Field geologic observation and sample collection strategies for planetary surface exploration: Insights from the 2010 Desert RATS geologist crewmembers, *Acta astronautica*, vol. 90, no. 2, pp. 344355, 2013, doi: 10.1016/j.actaastro.2011.10.015.

[5] K. Young and T. Graff, Planetary Science Context for EVA, in *NASA EVA Exploration Workshop*, 2020, p. 40.

[6] A. Kanelakos, Artemis EVA Flight Operations - Preparing for Lunar EVA Training & Execution, in *NASA EVA Exploration Workshop*, 2020, p. 47.

[7] M. J. Miller, Decision Support System Development For Human Extravehicular Activity, Georgia Institute of Technology, 2017.

[8] M. Hiltz, C. Rice, K. Boyle, R. Allison, and M. D. Space, CANADARM: 20 YEARS OF MISSION SUCCESS THROUGH ADAPTATION.

[9] R. Dechter, I. Meiri, and J. Pearl, Temporal constraint networks, *Artificial intelligence*, vol. 49, no. 1-3, pp. 6195, 1991, doi: 10.1016/0004-3702(91)90006-6.

[10] T. Vidal and H. Fargier, Handling Contingency in Temporal Constraint Networks: From Consistency to Controllabilities, *Journal of experimental and theoretical artificial intelligence*, vol. 11, no. 1, pp. 2345, 1999, doi: 10.1080/095281399146607.

[11] G. Casanova, C. Pralet, C. Lesire, and T. Vidal, Solving dynamic controllability problem of multi-agent plans with uncertainty using mixed integer linear programming, *Frontiers in artificial intelligence and applications*, vol. 285, pp. 930938, 2016, doi: 10.3233/978-1-61499-672-9-930.

[12] M. D. Moffitt, On the partial observability of temporal uncertainty, *Proceedings of the national conference on artificial intelligence*, vol. 2, pp. 10311037, 2007.

[13] N. Bhargava, C. Muise, and B. C. Williams, Variable-delay controllability, in *IJCAI International Joint Conference on Artificial Intelligence*, 2018, vol. 2018-July, pp. 46604666. doi: 10.24963/ijcai.2018/648.

[14] L. Hunsberger, A faster execution algorithm for dynamically controllable STNUs, *Proceedings of the 20th international symposium on temporal representation and reasoning*, pp. 2633, 2013, doi: 10.1109/TIME.2013.13.

[15] L. Hunsberger, Efficient execution of dynamically controllable simple temporal networks with uncertainty, *Acta informatica*, vol. 53, no. 2, pp. 89147, 2016, doi: 10.1007/s00236-015-0227-0.

[16] M. Ingham, R. Ragno, A. Wehowsky, and B. Williams, The Reactive Model-based Programming Language, MIT Space Systems and Artificial Intelligence Laboratories, 2002.

[17] Stanford Artificial Intelligence Laboratory et al., Robotic operating system. May 2018.

[18] G. Daines, Commercial lunar payload services. NASA, Mar. 2019.

[19] Consultative Committee for Space Data Systems, CCSDS File Delivery Protocol (CFDP), no. July, p. 151, 2020.

[20] C. Campbell, Advanced EMU Portable Life Support System (PLSS) and Shuttle/ISS EMU Schematics, a Comparison, *42nd international conference on environmental systems*, pp. 118, 2012, doi: 10.2514/6.2012-3411.

[21] D. A. Coan, Exploration EVA System Concept of Operations Summary for Artemis Phase 1 Lunar Surface Mission, NASA, Houston, TX, 2020.

[22] M. A. Seibert, D. S. Lim, M. J. Miller, D. Santiago-Materese, and M. T. Downs, Developing Future Deep-Space Telecommunication Architectures: A Historical Look at the Benefits of Analog Research on the Development of Solar System Internetworking for Future Human Spaceflight, *Astrobiology*, vol. 19, no. 3, pp. 462477, Mar. 2019, doi: 10.1089/ast.2018.1915.

[23] M. J. Miller and K. M. Feigh, *Addressing the envisioned world problem: A case study in human spaceflight operations*, vol. 5. Cambridge University Press, 2019. doi: 10.1017/dsj.2019.2.

[24] M. J. Miller, K. M. McGuire, and K. M. Feigh, Information flow model of human extravehicular activity operations, *Ieee aerospace conference proceedings*, vol. 2015-June, 2015, doi: 10.1109/AERO.2015.7118942.

[25] M. J. Miller, K. M. McGuire, and K. M. Feigh, Decision Support System Requirements Definition for Human Extravehicular Activity Based on Cognitive Work Analysis, *Journal of cognitive engineering and decision making*, vol. 11, no. 2, pp. 136165, 2017, doi: 10.1177/1555343416672112.

[26] A. Sehlke *et al.*, Requirements for Portable Instrument Suites during Human Scientific Exploration of Mars, vol. 19, no. 3, pp. 401425, 2019, doi: 10.1089/ast.2018.1841.

[27] P. H. Morris, N. Muscettola, and T. Vidal, Dynamic Control Of Plans With Temporal Uncertainty, 2001.

[28] N. Bhargava, C. Muise, T. Vaquero, and B. Williams, Delay Controllability : Multi-Agent Coordination under Communication Delay, Massachusetts Institute of Technology, Cambridge, MA, 2018.

[29] N. Bhargava, Multi-Agent Coordination under Limited Communication, Massachusetts Institute of Technology, 2020.

[30] R. Bellman, ON A ROUTING PROBLEM, no. 1.

[31] P. Morris, A structural characterization of temporal dynamic controllability, *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)*, vol. 4204 LNCS, pp. 375389, 2006, doi: 10.1007/11889205_28.

[32] P. Morris and N. Muscettola, Temporal dynamic controllability revisited, *Proceedings of the national conference on artificial intelligence*, vol. 3, pp. 11931198, 2005.

[33] N. Bhargava, C. Muise, T. Vaquero, and B. Williams, Managing communication costs under temporal uncertainty, in *IJCAI International Joint Conference on Artificial Intelligence*, 2018, vol. 2018-July, pp. 8490. doi: 10.24963/ijcai.2018/12.

[34] L. Hunsberger, Fixing the semantics for dynamic controllability and providing a more practical characterization of dynamic execution strategies, *Time 2009 - 16th international symposium on temporal representation and reasoning*, pp.

155162, 2009, doi: 10.1109/TIME.2009.25.

[35] P. Virtanen *et al.*, SciPy 1.0: Fundamental algorithms for scientific computing in python, *Nature methods*, vol. 17, pp. 261272, 2020, doi: 10.1038/s41592-019-0686-2.

[36] F. Pedregosa *et al.*, Scikit-learn: Machine learning in Python, *Journal of machine learning research*, vol. 12, pp. 28252830, 2011.

[37] J. D. Hunter, Matplotlib: A 2D graphics environment, *Computing in science & engineering*, vol. 9, no. 3, pp. 9095, 2007, doi: 10.1109/MCSE.2007.55.

[38] B. C. Williams, M. D. Ingham, S. H. Chung, and P. H. Elliott, Model-based programming of intelligent embedded systems and robotic space explorers, *Proceedings of the ieee*, vol. 91, no. 1, pp. 212236, 2003, doi: 10.1109/JPROC.2002.805828.

[39] B. C. Williams and R. J. Ragno, Conflict-directed A* and its role in model-based embedded systems, *Discrete applied mathematics*, vol. 155, no. 12, pp. 15621595, 2007, doi: 10.1016/j.dam.2005.10.022.

[40] P. Kim, B. C. Williams, and M. Abramson, Executing reactive, model-based programs through graph-based temporal planning, *Ijcai international joint conference on artificial intelligence*, pp. 487493, 2001.

[41] G. Kiczales, J. Des Rivières, and D. G. Bobrow, *The art of the metaobject protocol*. Cambridge, Mass: MIT Press, 1991.

[42] D. Ongaro and J. Ousterhout, In search of an understandable consensus algorithm, *Proceedings of the 2014 usenix annual technical conference, usenix atc 2014*, pp. 305319, 2014.

[43] L. Lamport *et al.*, The Part-Time Parliment, *Acm transactions on computer systems*, vol. 16, no. 2, pp. 373386, 1998, doi: 10.1145/568425.568433.

[44] J. L. Stedl, Managing Temporal Uncertainty Under Limited Communication: A Formal Model of Tight and Loose Team Coordination, Massachusetts Institute of Technology, 2004.

[45] A. Bit-Monnot, M. Ghallab, and F. Ingrand, Which contingent events to observe for the dynamic controllability of a plan, *Ijcai international joint conference on artificial intelligence*, vol. IJCAI-16, no. July, pp. 30383044, 2016.

[46] P. H. Morris and A. Bit-monnot, Dynamic Controllability with Single and Multiple Indirect Observations, in *ICAPS 2019 - Proceedings of the 29th International*

Conference on Automated Planning and Scheduling, 2019, p. 9.

[47] M. Fox and D. Long, PDDL2.1: An extension to PDDL for expressing temporal planning domains, *Journal of artificial intelligence research*, vol. 20, pp. 61124, 2003, doi: 10.1613/jair.1129.