Risk-Bounded Dynamic Scheduling of Temporal Plans

by

Andrew J. Wang

S.B. Aerospace Engineering with Information Technology (2011)
S.B. Electrical Engineering and Computer Science (2011)
M.Eng. Electrical Engineering and Computer Science (2013)
Massachusetts Institute of Technology

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2022

Author . . . . . .
Department of Electrical Engineering and Computer Science
August 12, 2022

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

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

Risk-Bounded Dynamic Scheduling of Temporal Plans

by

Andrew J. Wang

Submitted to the Department of Electrical Engineering and Computer Science
on August 12, 2022, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Real-world activities often have uncontrollable durations, which is a scheduling challenge for temporal planners, where despite uncertainty, we need to meet deadlines and coordination constraints. Missing these requirements can be costly, but it may be impossible to guarantee complete success. Instead, we aim to control the risk of scheduling failure, so we consider scheduling problems where activity durations are modeled with probability distributions. Then, by specifying a maximum allowable probability of failure, called a *chance constraint*, we gain access to solutions that are sufficiently safe. It is also known that reacting to duration outcomes throughout plan execution is key to avoiding failure. Thus, the goal of this thesis is to produce dynamic scheduling *policies* for chance-constrained temporal plans.

Our strategy is to build on prior art in chance-constrained static scheduling. First, we characterize more rigorously the reformulation of the original problem into that of risk allocation. This separates the probabilistic condition from the temporal conditions, but also introduces conservatism. Second, we generalize the static solution's conflict-directed hybrid algorithm to produce dynamic policies. Due to the chance constraint, we still employ nonlinear programming (NLP) to generate risk allocations, but now we leverage dynamic controllability (DC) algorithms to generate scheduling conflicts. However, those conflicts' resolutions are disjunctive constraints, which require combinatorial search and not just an NLP. So third, we map selected clauses into a form identical to that solved by the conflict-directed algorithm for static schedules. Our algorithmic architecture thus wraps the chance-constrained static solution within another layer of conflict discovery and resolution.

We evaluate our approach on lunar construction and car-sharing scenarios, which exemplify real-world complexity in coordinating parallel threads. We demonstrate that moving from chance-constrained static to dynamic policies dramatically increases the problem sizes we can schedule by at least 10 times. Additionally, our strategy for reallocating risk, based on discovered conflicts, solves an additional 10% of the benchmark scenarios over that achieved by uniform risk allocation. Finally, we show that our conflict-directed approach's runtime is an order-of-magnitude faster than solving a full encoding.

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

Acknowledgments

I joined the Model-based Embedded and Robotic Systems (MERS) group at a time when there was growing interest in reasoning about risk to ensure the safety of plans. My thesis advisor Prof. Brian Williams identified the need to extend such reasoning to temporal goals. And so, I would like to thank him for giving me the opportunity to work on the problem of chance-constrained scheduling.

The core of my algorithmic approach is heavily influenced by Brian's work in conflict-directed search. This theme has proven to be fundamental and enduring across our group's output. Thus, it has been meaningful to witness its broad power and applicability. In a way, this thesis can be viewed as an exercise in instantiating conflict-directed search to solve chance-constrained scheduling.

I have also absorbed much from observing Brian's teaching. He has always striven to be clear about the problem statement at hand and what need it is addressing. Additionally, he values expressing upfront the intuition behind algorithms and the key insights for tackling computational challenges. Brian has granted me multiple opportunities to practice these principles through guest-lecturing. These experiences have improved my own understanding of the state-of-the-art in non-probabilistic scheduling, which my work depends on.

Next, I greatly appreciate the perspectives of my thesis committee members Prof. Leslie Kaelbling and Prof. Armando Solar-Lezama, who are brilliant and experienced computer scientists outside of my immediate sphere. Interacting with both undoubtedly improved the presentation of my problem statement and algorithmic approach. Leslie, who is a roboticist, understands the planning community and literature well. Thus, our discussions were a good litmus test of whether the key insights of my work were well-specified for that audience. Armando's specialty is program synthesis, and although it is a distinct field from planning and scheduling, it does utilize shared concepts from constraint programming. It was illuminating and a great pleasure to receive insightful questions, from someone working in a different domain, about the structure and semantics in my work.

I have interacted with many fabulous labmates in the MERS group, including visitors. I would like to thank them all for contributing to my experience. In alphabetical order, they

are: Ben Ayton, Sara Bernardini, Nikhil Bhargava, Jake Broida, Sean Burke, Larry Bush, Rich Camilli, Stanley Cen, Jingkai Chen, Albert Chu, Sylvia Dai, Charles Dawson, Matt Deyo, Shannon Dong, Zach Duguid, Bobby Effinger, Simon Fang, Meng Feng, Enrique Fernandez, Carlos Forster, Wesley Graybill, Andreas Hofmann, Sungkweon Hong, Cyrus Huang, Ashkan Jasour, Erez Karpas, Boris Katz, Alex Klein, Spencer Lane, Sang Lee, Steve Levine, Shane Lyons, Nan Ma, Angelos Mallios, Lukas Missik, Christian Muise, Edward Nguyen, Johannes Norheim, Hiro Ono, Matt Orton, Viraj Parimi, Nick Pascucci, James Paterson, Amy Phung, Cameron Pittman, Jonathan Raiman, Marlyse Reeves, Paul Robertson, Pedro Santana, Yuki Sato, Shawn Schaffert, Howie Shrobe, Ameya Shroff, Szymon Sidor, Ellie Simonson, Delia Stephens, Dan Strawser, Yui Sujichantararat, Eric Timmons, Claudio Toledo, Tiago Vaquero, Allen Wang, David Wang, Maxwell Wang, Ethan Weber, Tesla Wells, Zhutian Yang, Peng Yu, and Yuening Zhang.

In particular, certain labmates have contributed significantly to my understanding of the ideas in this thesis, as well as how to implement them. I wish to recognize them as follows, in roughly chronological order:

Simon Fang collaborated with me to formulate our risk allocation approach to chance-constrained scheduling. He initially developed a pure nonlinear programming solution, but he also has a deep understanding of the structure of conflicts, and was thus an excellent sounding board for my thesis.

Peng Yu worked on the problem of temporal relaxation, and my discussions with him were my first introduction to the concept of temporal conflicts. Peng also jump-started my reading of the literature on dynamic controllability, provided the initial implementations of such algorithms, and demonstrated the possibility of benchmarking on public transit scenarios.

David Wang was indispensable in introducing me to the underlying mechanics of STN scheduling algorithms. He is the author of the temporal consistency code that underpins my implementation of strong controllability. In addition, he helped me understand the role of scheduling within the context of temporal planning. It was a pleasure to work with him when I provided an early implementation of my Rubato algorithm to use as a subsolver in his planner tBurton.

Eric Timmons has left a broad influence on my work. A talented systems builder, Eric wrote many of the Lisp libraries that I depend on, including temporal network data structures, a front end to numeric solvers (co-authored with Enrique Fernandez), and foreign function interfaces to those solvers. He has modernized the group's repositories and software environment with astounding complexity and grace. And being able to access his mental encyclopedia of Lisp has been a boon. But not only does Eric wield computer wizardry, he also possesses deep knowledge of the constraint programming algorithms that lie at the core of our group. Like with David, I've immensely enjoyed speaking with Eric about our respective research and their larger contexts.

Steve Levine, who is adept at simplifying concepts, wrote a course tutorial on conflict-directed search that helped me grasp the key ideas. He also graciously acted as a sounding board and note-taker when I was studying the literature on dynamic controllability. Steve's own work dealt with dispatching, which includes scheduling decisions, so he knew the lingo well enough to reflect back my conceptions in ways I could easily verify.

Nikhil Bhargava gave me an initial overview of Paul Morris's $O(n^3)$ dynamic controllability algorithm, and he authored an implementation of it that I rely on. His lightning-fast grasp of the nuances in controllability theory meant he was a reliable check for my ideas, and he provided keen observations on which I could build my understanding.

Yuening Zhang studies scheduling in multiagent contexts, and thus offers valuable perspectives on issues surrounding controllability. I've greatly enjoyed sharing my ideas with her, and appreciated her always insightful, probing questions ranging from motivation to mechanics. Yuening has also been an instrumental collaborator in integrating our scheduling algorithms within the constraint programming software framework that Eric supplied.

Cameron Pittman, one of our latest additions, brings operational knowledge from his experience and continued relationships with NASA. It's inspiring to have that connection with real missions, and the lunar construction scenarios in my benchmarks are adapted from his work.

Lastly, I would like to thank the following labmates for offering feedback on my thesis document and my defense presentation: Jingkai Chen, Simon Fang, Sungkweon Hong, Cyrus Huang, Cameron Pittman, Marlyse Reeves, and Yuening Zhang.

Contents

Chapter 1

Introduction

Reasoning about time is an integral component of intelligent planning, especially in modern society, which is governed so much by schedules and coordination. We individuals are constantly planning *when* to do things and *how long* they should take. On the small scale, we organize our days according to when we need to meet others, how long it takes to travel between locations, and when we simply have time to ourselves. Along the way, time needs to be made for eating meals, getting enough sleep, and other essential health-maintaining tasks. On the large scale, much of industry and commerce revolves around timed deliverables, ranging from days to years in advance. Recent years, for instance, have demonstrated the effects of delays propagating through our interconnected supply chains[18] [62].

Generally, there are two opposing factors that make scheduling a challenge. First, there are criteria like deadlines or other coordination constraints that define what it means for a plan to temporally succeed. That is, these are the temporal *requirements* on the plan's execution. Then, there are the plan's activities themselves, each of which requires a certain *duration* to complete. In other words, the execution of activities is what determines whether the requirements are satisfied.

Sometimes, a scheduling challenge arises simply because the plan is very large, and much work is required to calculate a schedule. More often, though, activities in real life pose difficulties because their durations are not fully controllable by whoever is executing them. For instance, most forms of travel between a source and destination are subject to various disturbances, such as traffic and weather. And anyone who has ever tried to follow new

instructions, be it for a recipe or for assembling furniture, has also experienced frustrating delays. In these cases, prior to plan execution, there could be considerable uncertainty about the actual durations, which would only be known after each activity completes.

This uncertainty threatens our ability to guarantee that the requirements will be satisfied by the plan's execution, and thus poses an important scheduling problem. One technique for addressing this is to observe and respond to the durations' outcomes online during execution. This sort of dynamic reasoning is second-nature to us humans, as we constantly adjust our daily routines on-the-fly. It helps avoid failure by preserve flexibility until the last possible moment. However, to have guarantees on correctness, such reasoning typically needs to be tailored to the particular plan at hand. Therefore, a complementary technique is to analyze the plan's structure and requirements offline. By considering all possible outcomes in advance, we can develop contingencies that are scripted to varying degrees. Traditional space operations, for instance, undergo heavy scripting due to the high stakes.

The difference between these two techniques is reminiscent of the distinction between reactive and deliberative control for robotics and other autonomous systems [55]. While dynamic responses to the current state of plan execution usually scale well, hedging against all possibilities becomes impractical for large and complex plans. The key insight, though, is that extreme outcomes are so unlikely that they can be safely ignored. This is a reasonably strategy that we adopt in our everyday tasks. When we ride the subway or hail a taxi, we assume that we'll reach our destination within a fraction of an hour rather than a fraction of a day. If it's the latter, then some failure exogenous to the plan has occurred, and we adapt by making alternative plans on-the-fly.

While we are comfortable in our daily lives with ignoring the possibility of extreme durations, professional applications of scheduling often demand stronger guarantees. Namely, when such assumptions are made, the *risk* of doing so needs to be quantified, in order to justify the massive expenditures. For instance, if a lunar rover, costing hundreds of millions, is sent to explore inside a crater where sunlight does not reach for long periods, every meter it travels away from the rim is a meter it will have to double back on. Given a deadline due to limited battery life, the rover must make a calculated risk for how far inside it can safely explore. Without making such a calculation, we would typically resort back

to ultra-conservative scripted behavior, and even possibly forego the opportunity to explore the crater.

To address the problem of quantifying scheduling risk, this thesis considers temporal plans with probabilistic durations. That is, some activity durations are modeled as probability distributions, from which Nature selects the outcome. The key value in doing so is that it becomes well-defined to specify the risk of failing to satify the temporal requirements during execution. By specifying acceptable risk, then, we avoid the ultra-conservative criteria of handling *every* duration outcome. Instead, we open up a spectrum of scheduling solutions that becomes wider with increasing tolerance of risk. In effect, this formulation of the scheduling problem allows us to tune our scheduling solutions to more likely duration outcomes.

Prior work in risk-bounded scheduling [20] [67] [69] [68] produced static schedules, which lacks the valuable reactivity of online decision-making. This thesis thus generalizes that work to produce dynamic scheduling *policies*. Our first contribution is to improve the specificity of the problem statement with respect to outputting such policies. We then offer a more rigorous treatment of the algorithmic approach, clarifying the relationship between various spaces of mathematical objects. This facilitates our extending the algorithm to encompass the necessary concepts for solving the dynamic version of the risk-bounded scheduling problem.

The rest of this chapter presents an overview of the problem this thesis solves and its approach for doing so. Section 1.1 examines the need for risk-bounded scheduling by presenting two scenarios, one everyday and the other professional. Section 1.2 then lays out the major claims of this thesis and maps them to the respective chapters. Finally, Section 1.3 provides an executive summary of our problem and algorithmic approach.

1.1 Need for managing scheduling risk

To highlight the importance of reasoning about temporal risk, we present two scenarios that illustrate the role of such reasoning within larger planning contexts. The first is familiar to anyone who has ever taken public transit and needs to deal with the uncertainty of

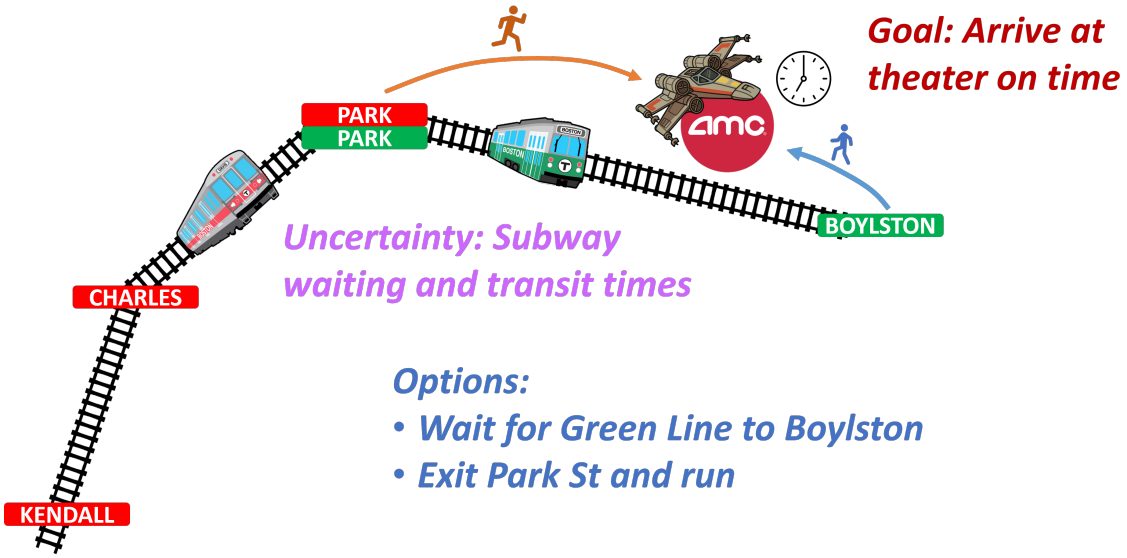

Figure 1-1: Catching the subway from Kendall to see a movie at the AMC in Boston Common is an experience shared by many students at the 'Tute. Typically, one transfers to the Green Line at Park Street and gets off at Bolyston, which is closer to the theater. However, if the transfer at Park is running late, then it might be safer to make a dash at street level, rather than wait.

transferring vehicles. The second is based on an actual situation faced by the Woods-Hole Oceanographic Institute (WHOI) when they were simultaneous operating multiple robotic ocean exploration vehicles. In both cases, we are faced with alternative planning choices, each of which result in a range of temporal outcomes that are tied to physical consequences.

1.1.1 Scheduling in everyday activities

Here along the banks of the Charles River, many students have experienced moviegoing at the AMC theater in Boston Common at some point. A popular mode to arrive there is to take the Red Line subway from Kendall Square into Boston. At the Park Street station, we transfer to the Green Line, and one stop later, we are at Bolyston station, which is right in front of the theater. This nominal plan is shown in Figure 1-1.

This plan requires us to wait twice for an arriving train, first at Kendall and then at Park Street. Each wait introduces uncertainty into our total travel time. Additionally, the travel times between stations plus the dwelling time at the intermediate Charles/MGH station are also out of our control. The uncertainty associated with these is typically less than

that of waiting for a train. However, if consecutive trains have not been maintaining their proper headways, then in an effort to restore them, the drivers may be forced to delay their departures or even stop in the middle between stations.

What this amounts to is that when we pull into Park Street on the Red Line, we need to consider the risk of waiting for a Green Line train to arrive (or more accurately, when it will actually depart). If we are trying to make it to a 7:00 p.m. showing, and it's already 6:50 p.m. at Park Street with no transfer in sight, we may be better off exiting to street level there and jogging the two blocks to the AMC. However, if we are lucky and a train has just pulled into Park Street as we enter the Green Line platform, there is a good chance that it will leave within a couple minutes, and thus arrive at Boylston with time to spare.

This scenario demonstrates that we are constantly assessing the risk of missing the start of the showing[1], since the beginning of the plan at Kendall and especially at the critical juncture of Park Street. Furthermore, we are reacting to the duration outcomes of each leg in the trip, and using that information to decide whether to continue with the original plan of riding to Boylston, or to switch to the alternate plan of exiting at Park. In both cases, we regain control of the duration for the last leg, where we can decide how fast to walk or run from the station into the theater.

Typically, the first decision, which is a discrete choice between alternative plans, would be classified as a planning decision, while the second one of controlling our own speed would be considered a scheduling decision. This thesis focuses on providing the latter functionality. However, the planning decision still rests on whether successful scheduling can be provided in either case. Therefore, the reasoning about risk that goes into our scheduling decisions is equally applicable in the larger context of planning.

The last point about this example is that we humans are accustomed to making such risk assessments on the fly, based on our previous experience with public transit. However, this is a fuzzy process and prone to mistakes as the plans increase in complexity. Instead, one could envision a digital assistant with statistical knowledge of waiting and transit times for the subway. Such empirical distributions could be used to quantify the actual risk of

[1]Actually, we might only miss some previews. However, if we're traveling with a large group, showing up late reduces the chances of finding good seats.

missing your deadlines, and thus better inform your decisions. Imagine, then, the leverage you'd hold over your friends the next time you argue inside Park Street station about whether to wait for the next train or to make a dash for it outdoors.

1.1.2 Scheduling in professional scenarios

While making bets with your friends is relatively harmless, scheduling risk can be a major factor in professional scenarios where the stakes are much higher. Figure 1-2 depicts an actual ocean exploration scenario where scientists onboard a ship are studying the seafloor using a tethered remotely operated vehicle (ROV), while an autonomous underwater vehicle (AUV) is surveying nearby waters. At some point, the AUV was meant to return to the ship for recovery, but due to a navigational error, the AUV overshot, didn't recognize it, and continued in a beeline. The danger is that its trajectory eventually crosses a shipping lane, where it risks collision.

The ship's crew thus faces the following decision. They could gun the ship towards the AUV and try to capture it before it hits the shipping lane, while reeling out the tether for the ROV. Or, they could reel in the ROV first, and then chase after the AUV. Based on vehicle cost alone, they might prioritize the ROV and choose the second option, but the AUV might have traveled so far by then that they can't catch up with it in time. Therefore, it's worth considering the first option of prioritizing the AUV. The risk there, though, is if the distance between the ship and the ROV exceeds the total length of the tether, then the tether breaks, we lose communication with the ROV, and we will have conduct an expensive follow-up ROV recovery after retrieving the AUV. A middle ground, then, would be to chase after the AUV first, but if we are running out of tether, then we switch to reeling it in, and then continue to retrieve the AUV.

While this scenario is ultimately about meeting spatial targets, the range of speeds for each vehicle relates the distance it travels into a range of possible durations. In theory, then, we should be able to calculate how long it would take the ship to catch up with the AUV in both scenarios, and decide which plan leaves more overall margin. Unfortunately, there is the complicating factor of ocean currents, which vary considerably in all three dimensions

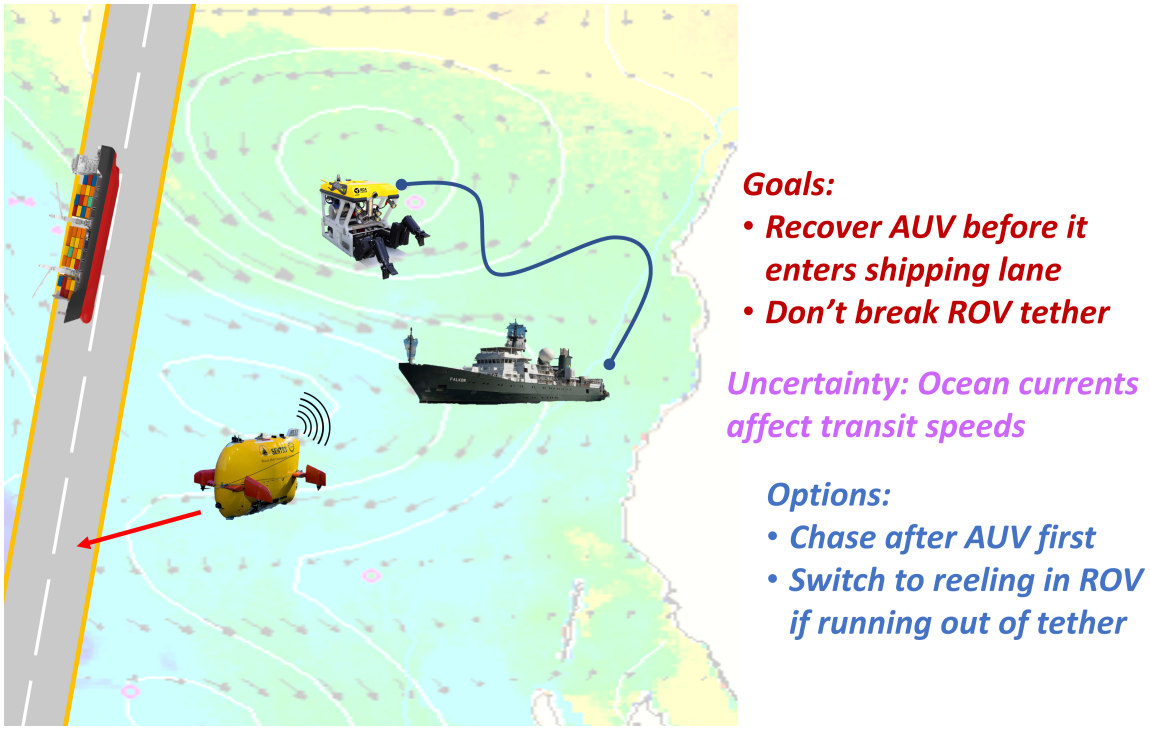

Figure 1-2: Faced with a rogue AUV while conducting a tethered ROV mission, the ship's crew has to decide which vehicle to recover first. If the AUV is let loose for too long, it may cross into a busy shipping lane. However, it would also be expensive to break the tether and have to recover the ROV later. The temporal uncertainty in each scenario comes from the ever-shifting ocean currents, which affect the transit times of all three vehicles.

and are notoriously hard to predict. Since the ship has the most powerful engines, its transit durations are least affected by such currents, and surface ones at that. Conversely, the energy-efficient AUV has the weakest propulsion, so its true distance from the ship over time has fast-growing uncertainty. An accurate analysis of the situation must therefore take into account this uncertainty for each vehicle, and consider the potential delay it introduces in achieving the necessary rendezvous.

This scenario faces similar reasoning challenges as the moviegoing example. First, the safety of either alternative is heavily influenced by the actual durations it takes to complete the required actions. And second, those durations are difficult to estimate, but their distributions could be maintained by a planning system, and thus contribute towards a quantified assessment of the scheduling risk. In this case, historical data could be combined with models of the vehicle dynamics and forecasts of the ocean currents to yield distributions on the transit times between any two points for any vehicle.

In both the moviegoing example and this AUV retrieval situation, the scenario was small enough that the humans in charge could personally intuit the risks and decide based on that. However, suppose we wanted to simultaneously deploy a large number and wide variety of robotic vehicles to explore in parallel. Such scenarios would greatly advance the investigative abilities of oceanographers, but also require a degree of coordination that would overwhelm our intuition for assessing risk. Instead, automated risk analyses could be powerful tools for the human operators in charge, and thus enablers of such deployments.

1.2 Thesis claims and outline

In this section, we state in four main claims the technical contributions of this thesis to risk-bounded scheduling. Each claim is addressed by a subsequent chapter, ranging from Chapters 2 to 5.

The first claim pertains to the problem statement. In the non-probabilistic setting, many previous authors have contributed to specifying the problem of dynamic scheduling. Similarly, we found that the notion of risk-bounded scheduling, seemingly intuitive, actually rests on an intricate network of concepts. Our goal is to elucidate that network and to enrich

it with the work that went into defining dynamic scheduling.

Claim 1.1. *We extend the concept of chance-constrained scheduling to encompass dynamic scheduling. That is, we rigorously define what it means for a dynamic scheduling policy to satisfy a chance constraint.*

We use the term "chance-constrained" as a more precise synonym for "risk-bounded", since the problem statement actually contains a *chance constraint* that formally bounds the acceptable probability of failure. From this point forward, we will thus use the term "chance-constrained" when describing our problem.

Chapter 2 defines our chance-constrained scheduling problem, where the key concepts we introduce are those of probabilistic durations and chance constraints. In order for these to be well-defined, we rely on a foundational model of scheduling where the duration uncertainty is not probabilistic but rather set-bounded. This turns out to be important later in our solution approach. One concept we require, though, that prior work in scheduling does not make, is the distinction between temporal requirements and activity durations. Thus, we present upfront the set-bounded model, but now with that distinction built in.

In this presentation, we also address the full semantics of dynamic execution. The first reason is that these semantics automatically carry over into the chance-constrained setting. The second is that we explicitly show static schedules can be framed as a special case of dynamic policies. These two reasons combined mean that our problem statement is a proper generalization of prior art on chance-constrained static scheduling into dynamic scheduling.

With the problem in hand, we investigate the computational challenges in solving it, and present a crucial reformulation of it into a more tractable form. It turns out most prior work in probabilistic scheduling has used the same or similar reformulations. However, many have not explicitly acknowledged that, so Chapter 3 examines the structure of that reformulation, as well as its implications for soundness and completeness.

Claim 1.2. *We reformulate the chance-constrained problem into one of risk allocation, thereby decoupling the probabilistic requirement from the temporal constraints of the plan. We introduce more rigor than prior art does when defining this reformulation, and also in characterizing its inherent sources of incompleteness.*

As hinted above, this reformulation takes advantage of the well-developed theory for set-bounded scheduling. We enforce the chance constraint in a way that effectively ignores the unlikely extreme cases of duration outcomes. We call this *risk allocation*, and in doing so, we transform the probabilistic durations of our plan into set-bounded durations. The remaining constraints of the problem are then identical to that of set-bounded scheduling. There are two aspects to this that we develop further than prior art.

First, this reformulation implies that when executing plans with probabilistic uncertainty, we are actually employing scheduling policies that were designed for set-bounded uncertainty. However, by definition, the latter encompasses more potential duration outcomes than the former. Since a policy needs to be well-defined for any outcome, we actually need to extend any valid set-bounded scheduling policies we find into a probabilistic context. Fortunately, the concepts developed in Chapter 2 aid in this specification.

Second, the chance constraint enforcement strategy does rule out potentially valid policies that cannot be accessed with the set-bounded model. It can be argued that the policy space we give up would have been needed only for handling unlikely extreme outcomes. However, it is still important to document these losses. To do so, we systematically illustrate the relationships between the space of policies, the space of outcomes, and the space of risk allocations. This makes it easy to identify the various sources of incompleteness, which prior art had glossed over.

One of the major results of chance-constrained static scheduling is that a solving the reformulated problem via a *conflict-directed hybrid* approach achieves significant runtime gains [69]. Rather than encoding all the constraints, we iteratively discover conflicts from the set-bounded theory, and use them to improve our risk allocation. It is thus natural to ask whether those conflicts can be adapted to give information about producing dynamic policies.

Claim 1.3. *We adapt the conflict-directed hybrid approach of prior art to check for dynamic controllability conflicts, so as to produce dynamic policies in the end. We characterize the form of these conflicts' resolutions as disjunctive linear constraints in the risk allocation space.*

Chapter 4 gives a comprehensive treatment of how we leverage principles from conflict-directed search to solve our problem. We begin by showing that fully encoding all the constraints of our reformulated problem would be expensive, especially when searching for dynamic policies. Then, we develop a hybrid algorithmic solution that uses two black-box solvers to handle the two main conditions of the reformulated problem. A nonlinear programming (NLP) solver is needed to generate candidate risk allocation, because the chance constraint is inherently nonlinear. Then a controllability checker verifies whether a policy exists for the resulting plan with set-bounded uncertainty. If not, it returns a controllability conflict that further informs the NLP solver when generating subsequent risk allocations.

The key step in this architecture is to understand the form of the conflicts and how to resolve them. We develop the intuition to show that these conflicts are different depending on whether we are solving the static or dynamic variant of our chance-constrained scheduling problem. Namely, the dynamic controllability conflicts block out less of the risk allocation space, which matches our hypothesis that enabling dynamic scheduling decisions widens the space of feasible scheduling solutions. However, those conflicts' resolutions have a disjunctive form, which requires more general solvers than those specialized for NLP.

Rather than employ such a general-purpose solver, which is expensive, our insight is to reduce the problem of handling those conflicts into a series of subproblems without disjunctions. That is, those subproblems can be solved by existing algorithms for finding chance-constrained static schedules.

Claim 1.4. *We show that the combinatorial search required to handle the disjunctions creates subproblems that are identical in form to the static variant of our problem. We are thus able to frame chance-constrained static scheduling as a subproblem in our algorithm for producing dynamic policies, and consequently leverage that existing solution.*

Chapter 5 presents this insight and implements it by separating out three layers of interacting subproblems. The bottom two layers form our solution for finding chance-constrained static policies, and are functionally identical to our architecture in Chapter 4. Then, instead of replacing the NLP solver in the second layer, we add a third layer on top to

handle the branching on disjunctive conflict resolutions. Once we reach a leaf in our search tree, the selected disjuncts are ready to be processed by the second layer's NLP solver.

To further unify the algorithmic architecture, we show that all three layers share common principles of finding and using conflicts in their solution methods. Therefore, we distill these principles into a template algorithm that relies on conflicts from the layer below and returns its own conflicts to the layer above. We then present each layer's solution as an instantiation of this template, so the full solution can be understood as the composition of core principles across different levels of abstraction.

Beyond the four technical claims of this thesis, we survey the relevant literature and offer experimental results. Chapter 6 reviews how others have approached similar scheduling problems in probabilistic settings. Chapter 7 then presents benchmarks that demonstrate our generalization of risk-bounded static scheduling to dynamic policies allows us to access many more solutions. We also demonstrate the runtime advantages of our conflict-directed algorithmic approach. Finally, Chapter 8 summarizes the thesis and offers roadmaps for future work.

1.3 Approach in a nutshell

In this section, we walk through the problem statement and solution approach of this thesis. The goal is to illustrate the concepts intuitively, and we rely heavily on visual depictions. Most of the figures are adapted from subsequent chapters, where the concepts are more deeply developed. In the figures presented here, we include details for completeness, but only discuss the most salient points.

To define the notion of scheduling risk, we need to model activities' durations probabilistically. The modeling framework we use is called the *probabilistic simple temporal network (pSTN)*. Figure 1-3 depicts a pSTN that models a plan for making spaghetti. Each arrow, except the red one, represents an activity, and through events, which are instantaneous points in time, the activities link up to form threads, all moving forward in time to the right.

The key feature of a pSTN is that certain activities, indicated in dotted purple, have probabilistic durations. Once such an activity is initiated, Nature samples a duration for

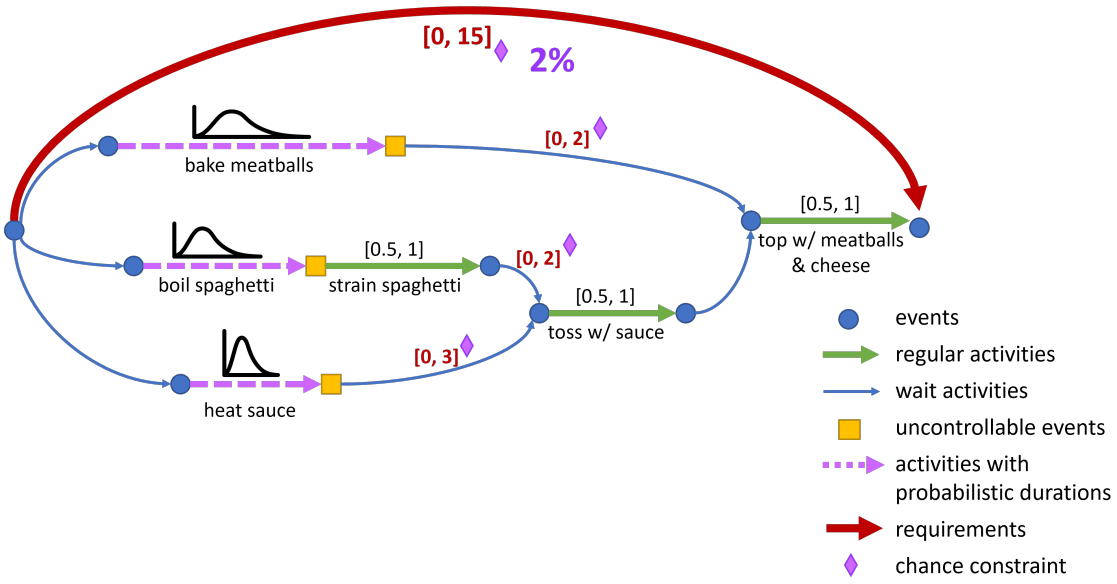

Figure 1-3: This is a pSTN that models a plan for making spaghetti. The cooking activities have uncontrollable duration, modeled probabilistically, while all the others are under the chef's control. Various temporal requirements are imposed throughout the plan, the main one being the 15 minute overall deadline. The goal is to satify those requirements with no more than 2% risk of failure.

it out of its distribution, and we as the plan executive do not find out until the activity completes with that duration. The remaining activities, in green and blue, can be controlled by us, within the time windows listed if any.

The final feature of a pSTN is the imposing of temporal requirements, indicated in red. In the figure, we have a 15-minute deadline for completing the entire plan. We also have local deadlines for how long we are willing to let the meatballs, spaghetti, and sauce sit around before we begin the tossing and topping activities. The requirements are what define the success criteria for the plan's execution. Therefore, they are the constraints on which we apply the notion of bounded risk.

The purple diamonds in Figure 1-3 represent a 2% *chance constraint* that is imposed on the requirements. This expresses the idea that across all possible executions of the pSTN, weighted by the probabilistic durations' outcomes, the likelihood of violating any one of the requirements may not be over 2%.

This definition of bounded risk relies on a clear notion of how a pSTN's execution is determined by a *scheduling policy*. Fortunately, we are able to borrow the definition

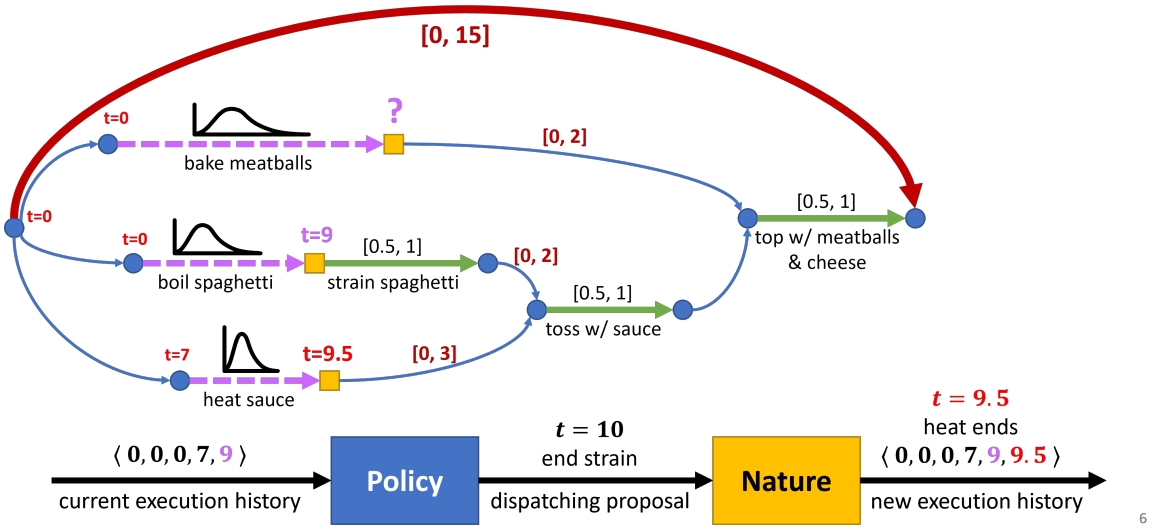

Figure 1-4: As an example of dynamic execution, here is a situation where we've started cooking all three ingredients, but only the boiling activity has completed at time $t = 9$. Our dynamic policy decides to finish the straining activity at $t = 10$. However, Nature decides that the sauce is done at $t = 9.5$. This gets recorded in the execution history, so now our policy has the option to respond by either straining at the same rate, or trying to finish it earlier.

of a dynamic policy for *simple temporal networks with uncertainty (STNUs)*, and apply it directly to pSTNs. Figure 1-4 illustrates how such a policy might behave in the middle of executing our spaghetti example.

This figure begins with an execution state where the meatballs are still baking in the oven, the spaghetti has been taken off the burner at time $t = 9$, and the sauce is still being heated. Thus, the current time is $t = 9$, and we've just begun straining the spaghetti. At this point, the policy may decide to strain for one whole minute, ending that activity at $t = 10$. However, we may find out at $t = 9.5$ that Nature has decided to terminate the heating sauce activity. [2] Hence this observation gives us the opportunity to reevaluate its $t = 10$ decision. We could decide to stay the course, or finish straining a little faster sometime in the 30-second window between $t = 9.5$ and $t = 10$.

In this scenario, there is no particular advantage to finishing faster, but this is the kind of flexibility that is key to dynamic execution. In other scenarios, we may wish to start or finish activities later in response to observed uncontrollable durations. This dynamic execution

[2] It is somewhat formal to say that Nature "terminates" this activity. In real life, this likely corresponds to us noticing that the sauce is bubbling vigorously, and we should take it off the heat.

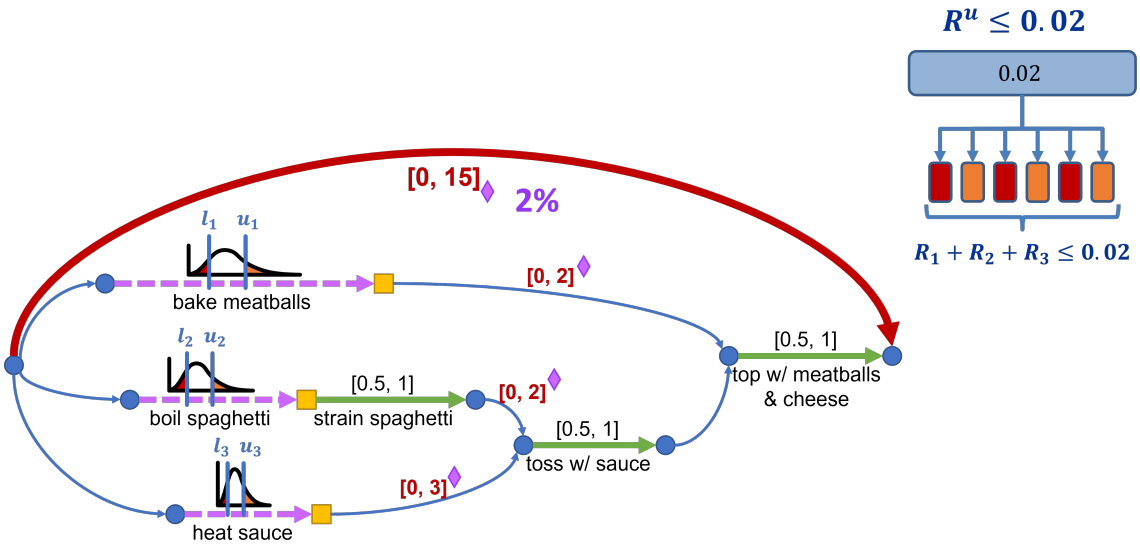

Figure 1-5: The core idea of our risk allocation reformulation is to impose $[l, u]$ interval bounds on each probabilistic duration. First, this allows us to enforce that the probability mass in the removed tails must not add up to more than the chance constraint's risk bound. Second, it maps the pSTN into an STNU, for which we can enforce controllability conditions.

mechanism thus completes our problem definition, namely, that given a chance-constrained pSTN (cc-pSTN), we seek dynamic policies that will satisfy the chance constraint.

As given, the cc-pSTN is rather intractable due to two main factors. First, we allow arbitrary probability distributions for the durations, so as the activities compose throughout the plan, it becomes difficult to evaluate the actual probability that any given requirement will be violated. Furthermore, this evaluation depends on the behavior of the policy we choose, and not just Nature's sampled outcomes. As Figure fig:nutshell-spaghetti-execution demonstrates, the policy space would be extremely rich and difficult to navigate.

To sidestep these difficulties, the first step in our solution approach is to reformulate the cc-pSTN into a more specific problem that follows the form of risk allocation. Figure 1-5 illustrates this being applied to our spaghetti scenario. Rather than consider probabilities densities across the entire space of outcomes, we ignore the extreme cases in the tails, and focus on the outcomes within an interval $[l, u]$ for each outcome. This allows us to decouple the probabilistic condition of the chance constraint from the temporal condition of satisfying the requirements, as follows.

First, we account for the probability mass in the tails that we ignore, and make sure that

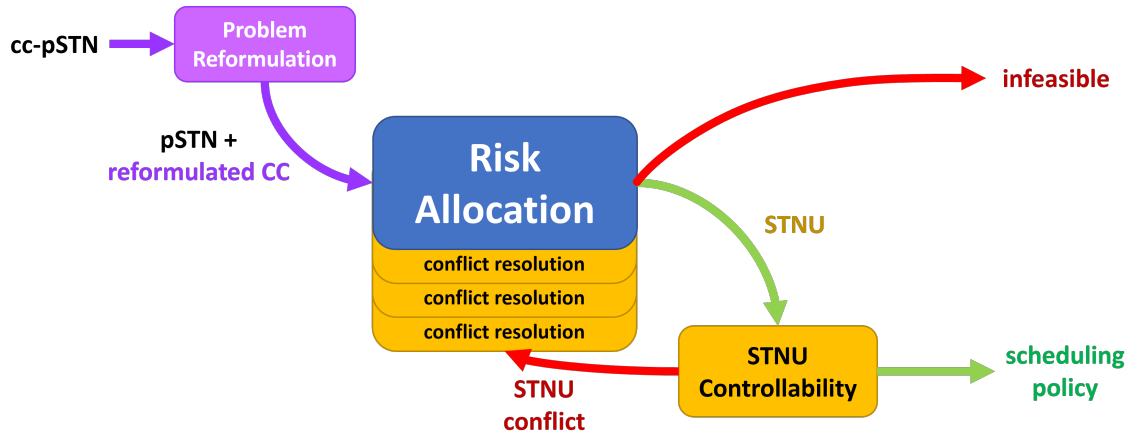

Figure 1-6: Our conflict-directed approach uses an STNU controllability checker to discover conflicts that inform subsequent rounds of generating risk allocations. This avoids having to encode all the controllability conditions at once. Eventually, either we will find a risk allocation whose associated STNU is controllable, or we will have collected enough conflicts that make the risk allocation problem infeasible.

it is less than the chance constraint's risk bound. We call this condition the reformulated chance constraint, and any valid solution based on ignoring those tails will thus respect the original chance constraint. To get such a solution, we note that the $[l, u]$ bounds convert our pSTN into an STNU, where the uncontrollable durations no longer have probabilistic uncertainty, but only set-bounded. Thus, we can leverage STNU controllability theory to express the deterministic condition that a policy exists for the converted STNU. In summary, our reformulated problem is to find $[l, u]$ bounds, otherwise known as a *risk allocation*, such that they satisfy the reformulated chance constraint *and* they result in a dynamically controllable STNU.

To solve the reformulated problem, we note that fully encoding the STNU controllability constraints would be expensive. Instead, we use a conflict-directed, hybrid approach that follows a generate-and-test paradigm for converging to a valid risk allocation. Figure 1-6 illustrates this algorithmic architecture.

After reformulating the original problem, we generate a risk allocation, using an NLP solver, *without* regard for the controllability conditions. Then, we *check* whether the implied STNU is controllable, which is a much faster operation than trying to solve for such an STNU. If it is controllable, then we've satisfied the necessary conditions. Otherwise,

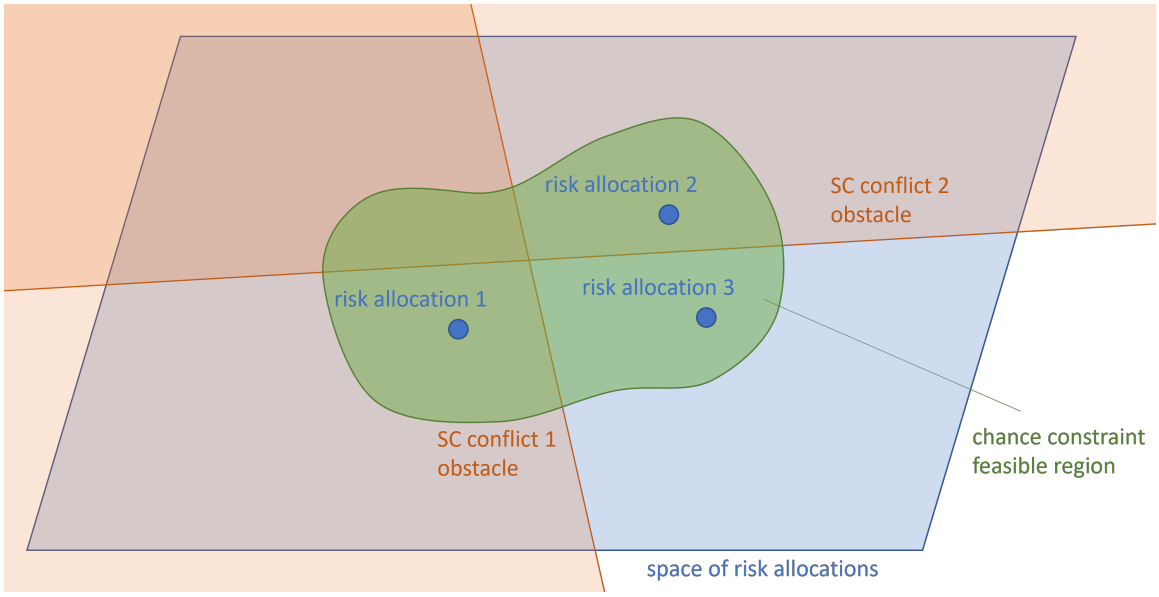

Figure 1-7: For static policies, the strong controllability conflicts we discover are hyperplane-bounded regions in the risk allocation space. With each risk allocation we generate, its associated conflict cuts down on the remaining feasible region of the chance constraint.

we extract a *conflict* from the controllability checker, and derive a resolution constraint that feeds back into the NLP solver for another candidate risk allocation. Thus, this approach is hybrid in the sense that we leverage two black-box solvers to address respective subsets of the full problem.

Intuitively, this algorithm replaces the full encoding of STNU controllability with an incremental collection of conflict resolutions. If we kept collecting them, we would eventually have an alternative formulation of the full encoding. However, with the chance constraint limiting us to a certain set of risk allocations, we likely only need to discover a few conflicts to find a valid solution, or otherwise prove infeasibility.

The key to this algorithm's operation, then, is to understand the structure of the conflicts and their resolutions. Although we are ultimately aiming for dynamic policies, it is helpful to first consider how we obtain static policies, since the conflict-directed approach is equally applicable in both cases. The only difference is that to obtain static policies, we check for STNU strong controllability (SC), whereas obtaining dynamic policies requires that we check for dynamic controllability (DC).

Figure 1-7 depicts how we navigate the space of risk allocations using strong control-

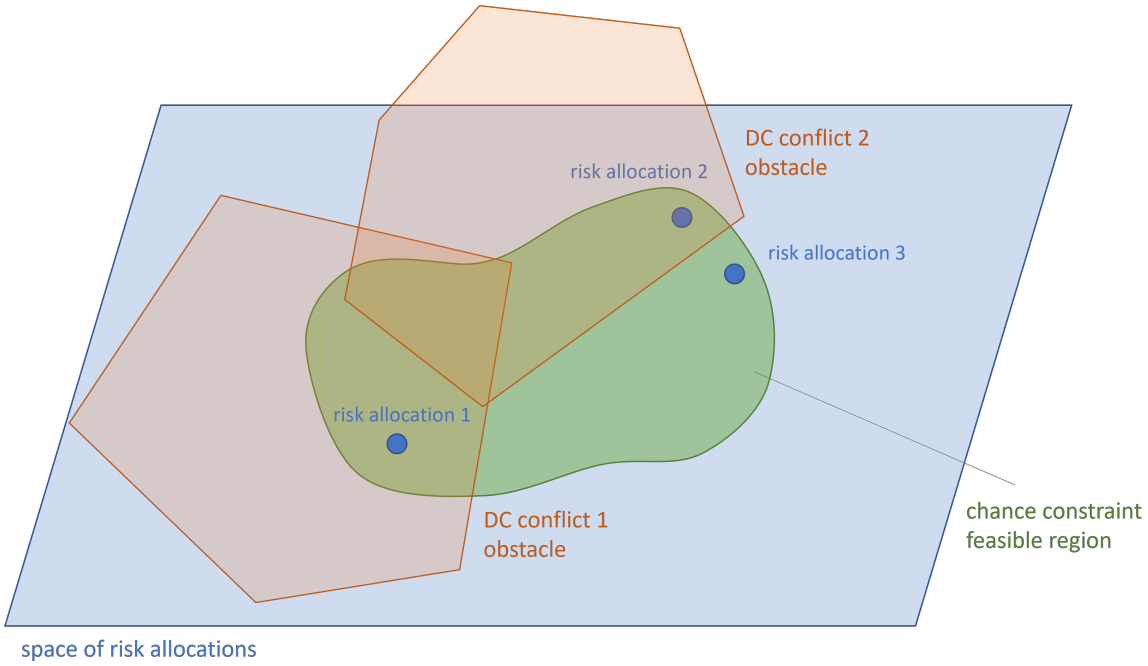

Figure 1-8: For dynamic policies, the dynamic controllability conflicts form convex polytope obstacles instead of half-volumes. Therefore, they also also trim the chance constraint region, but less aggressively, and hence are more likely to lead to a solution.

lability conflicts. Initially, we choose any risk allocation within the green feasible region specified by the reformulated chance constraint. If it results in an STNU that's not strongly controllable, then the conflict says that STNU has a negative cycle in its distance graph.

We don't show the distance graph here, but the key result is that we avoid making that cycle negative by deriving a linear "conflict resolution" constraint in the risk allocation space. This means a half-volume containing the original risk allocation is pruned away, and hence also cuts out part of the chance constraint feasible region. The conflict-directed algorithm thus repeats this process, generating risk allocations in the remaining green region, and cutting out portions that do not satify strong controllability.

When we replace the controllability check in Figure 1-6 with dynamic controllability, the conflicts that are returned are called *semi-reducible negative cycles (SRNCs)* in the STNU's distance graph. Without going into detail here, the result is that the SRNC-avoidance obstacle becomes a convex polytope instead of a half-volume, as shown in Figure 1-8. Algebraically, the conflict resolution constraints are now *disjunctions* of linear constraints.

The major consequence of using DC conflicts versus SC conflicts is that each conflict

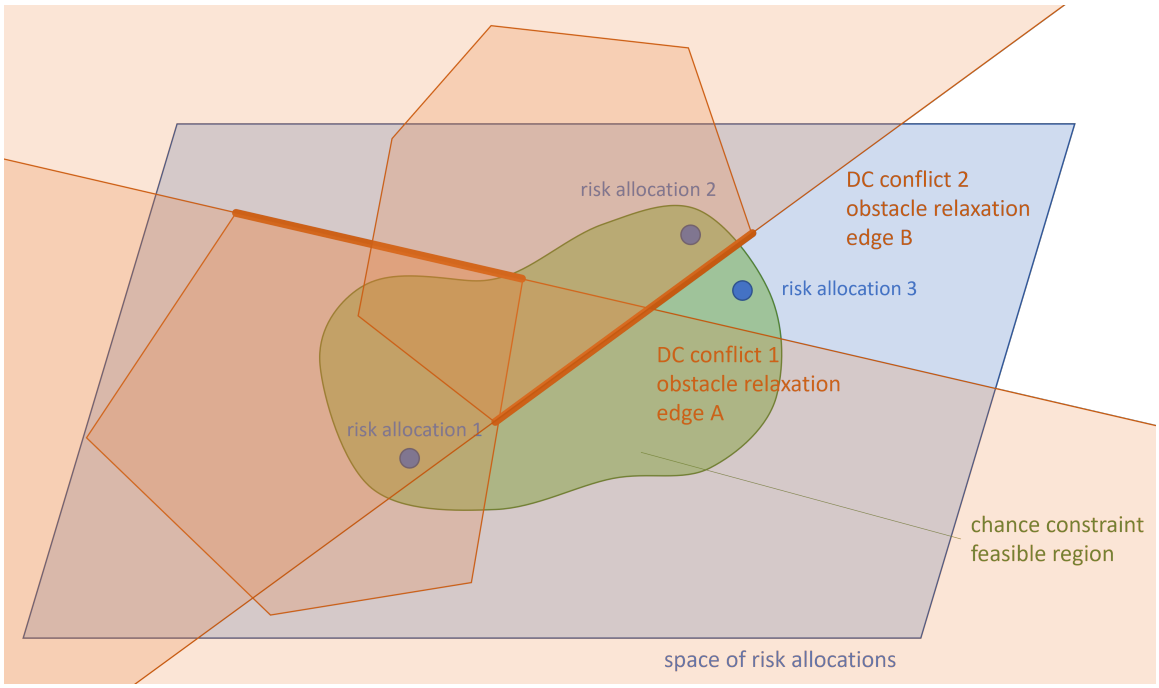

Figure 1-9: When we branch on the DC conflicts' facets, we relax the polytopes into half-volumes. This mimics the form of the SC conflicts' resolution constraints. Thus, once all DC conflicts have been branched on, we can use the same static policy algorithm, with minor adjustments, to find a risk allocation.

obstacle cuts out less of the risk allocation space, and hence potentially less of the chance constraint region as well. This is what accounts for the increased solution space when solving for dynamic policies. The difficulty, however, is that the NLP solver for the risk allocation step in Figure 1-6 isn't designed to handle disjunctive constraints. Swapping it out for another black-box solver than can, such as those that perform mixed-integer nonlinear programming (MINLP), is theoretically possible, but would be very expensive.

Instead, our key insight is that by branching on the linear disjuncts of each disjunction, we map the problem of solving dynamic conflict resolution into a series of linear programs (LPs) with the nonlinear chance constraint attached to each. This is exactly the form of constraint program that is solved by our NLP solver in the static case, and it is depicted in Figure 1-9.

We have the same two DC conflict obstacles as in Figure 1-8, but now we pick a single "facet" of each obstacle to branch on. Each facet corresponds to a linear disjunct. With facet A from obstacle 1 and facet B from obstacle 2 activated, they prune out half-volumes

just like the ones in Figure 1-7. In this case, we are left with a sliver of the chance constraint region, from which we select a third risk allocation.

Therefore, the strategy to solve for dynamic policies is to try all combinations of facets from each known DC obstacle. For each combination, we use the static policy algorithm to solve for a risk allocation, and with the modification of checking STNU *dynamic* controllability. Some combinations may completely prune the chance constraint region, so those are dead ends. Other combinations may yield STNUs that fail the DC check, in which case we discover new SRNC conflicts. Only when we've exhausted all combinations without finding a DC STNU do we return infeasible.

This strategy of branching over disjunctions can be abstracted into combinatorial search, where we map each disjunction into a discrete variable, and each disjunct into a value for that variable. The insight this abstraction brings is that the relationship between its search and problem it frames for the static policy algorithm mirrors the relationship between the NLP solver and the STNU problem it frames for the controllability checker. In both cases, a higher level is responsible for the entire problem, while relying on a subsolver to verify the remaining conditions. When the subsolver cannot generate a solution to the subproblem it's given, it returns a conflict for the higher level to resolve.

This insight leads us to unify our algorithms for generating chance-constrained static and dynamic policies. We identify layers of subproblems and construct algorithms for each. Each layer's algorithm follows a common template where it calls a third-party solver to handle the constraints that layer is designed for, and then frames a subproblem for the layer below to verify. Figures 1-10 and 1-11 depict this architecture for static and dynamic policies, respectively.

The main difference is that the dynamic version inserts a third layer between the common top level, which performs the problem reformulation, and the second layer, which is the entry point for the static algorithm. Minor modifications are also needed for the bottom two layers. First, the bottommost layer now returns SRNC conflicts instead of just plain negative cycles. However, a point of nuance is that under certain conditions, some SRNCs only have a single linear resolution, which is not disjunctive. Thus, the second layer can actually handle those resolutions and generate new risk allocations without going up to the

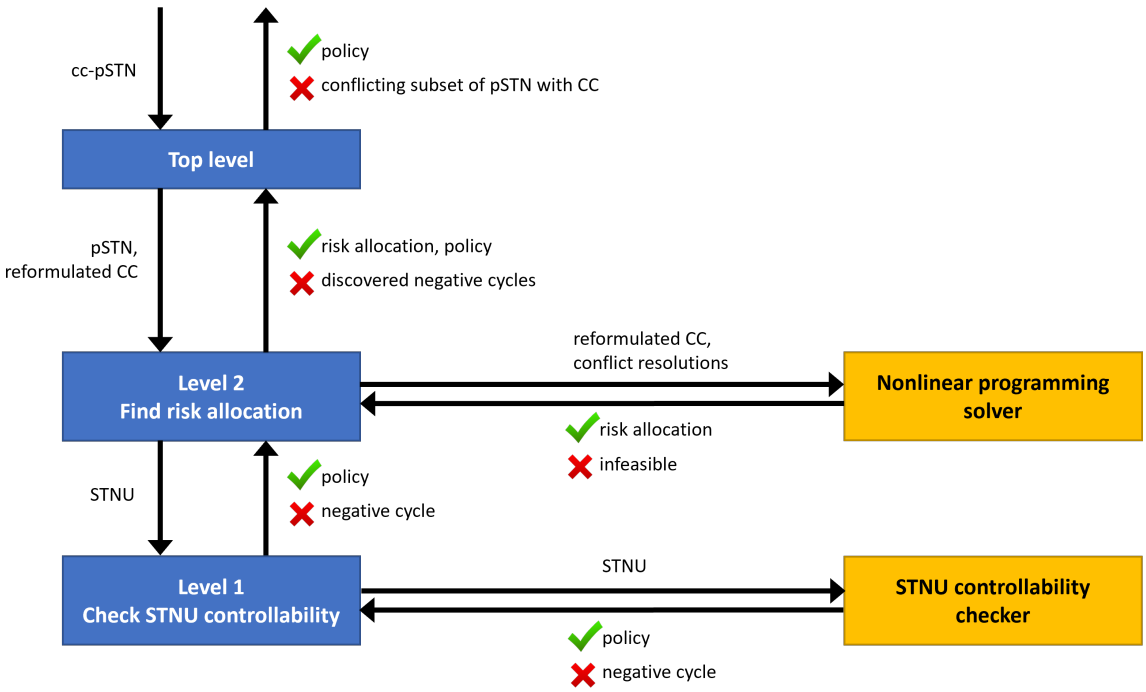

Figure 1-10: Our algorithm for producing chance-constrained static policies is composed of two interacting levels, underneath a top level which is responsible for the problem reformulation. Level 2 uses an NLP solver to generate risk allocations, while Level 1 checks the implied STNU for strong controllability.

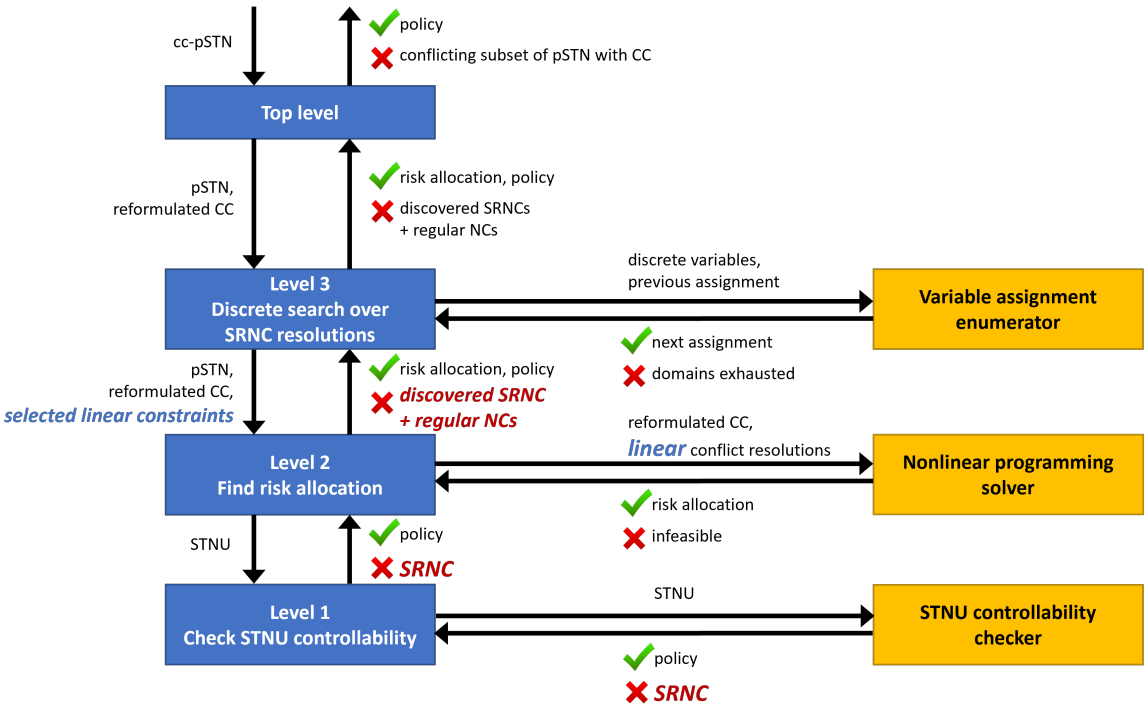

Figure 1-11: To produce chance-constrained dynamic policies, we incorporate a third level between the top level and Level 2 to perform combinatorial search over the polytope facets. These polytopes are generated via a modification to Level 1 to check for dynamic rather than strong controllability.

third layer. Only when faced with a disjunctive constraint does the second layer return that as a conflict up to the third layer for branching.

In summary, this section has presented a fast-paced, illustrated overview of this thesis's technical contributions. We began by showcasing the main features of cc-pSTN problem and that of finding a dynamic policy. Then we reformulated the problem into a more tractable risk allocation form. The structure of the reformulated problem led us to design a conflict-directed, hybrid algorithm, where STNU controllability conflicts inform subsequent rounds of risk allocation. The final step is to recognize that DC conflicts can be resolved by branching on them in forms that mimic the SC conflict resolutions. This leads to a shared algorithmic solution, where finding a chance-constrained dynamic policy is implemented as a layer of combinatorial search over the algorithm for chance-constrained static policies. Subsequent chapters develop each of these points thoroughly and in sequence.

Chapter 2

Problem Statement for Risk-Bounded Scheduling

Chapter 1 had expressed the need to model activities' durations with probability distributions, so that large, complex plans could be scheduled with bounded, quantifiable risk. The goal of this chapter is thus to formally define the scheduling problem for plans with probabilistic durations.

A key feature of this problem is the *probabilistic* condition for success. Unlike problems based on non-probabilistic models, where the objective is to handle *all* possible uncertain outcomes, our objective will be to handle enough outcomes such that we have a *sufficiently high probability* of successfully scheduling the plan. I capture this notion of sufficiency with a *chance constraint*, and I argue why this is preferred to simply minimizing the risk of failure.

The second key feature of my problem is that it targets dynamic execution of the plan. Previous work [67] [20] [69] had addressed a similar version of the problem that allowed only for static execution. This limitation prevents us from reacting to duration outcomes as they occur, and thus severely limits the solution space. In this chapter, I build up the concept of a *scheduling policy* to support dynamic execution. I then show how it also encompasses static execution as an edge case.

The notion of dynamic execution has been well-studied for a non-probabilistic plan model called the *simple temporal network with uncertainty (STNU)* [38] [33] [40]. This

model captures most of what we want *except* the probabilistic duration and the consequences it entails for the problem statement. Therefore, I build our model, the *probabilistic simple temporal network (pSTN)* by extending/modifying the STNU's semantics.

Section 2.1 gives a thorough review of the STNU model and the form of its scheduling policies. This allows me to rigorously define, in Section 2.2, the pSTN model and the *chance-constrained*-pSTN (cc-pSTN) scheduling problem as precise extensions to their STNU counterparts. Although Section 2.1 recaps much existing work, from the beginning I present it with a key innovation to support my cc-pSTN definition. Namely, since the chance constraint is a probabilistic condition on the plan being *properly scheduled*, I separate out the notion of a *temporal requirement* from the activity durations in the plan. With the exception of [69], prior work does not make this distinction, but I argue that this is necessary for the notion of a chance constraint to be semantically sound in the context of plan execution.

2.1 Review of STNUs

Temporal plans are composed of activities that have flexible duration. This means during execution, an activity's duration could fall anywhere within some domain. For instance, when I brush my teeth in the morning, it takes between one and three minutes, depending on how thorough I wish to be. By composing multiple such activities (e.g., making the bed, eating breakfast, doing dishes, etc.), and then imposing some constraints, such as a one-hour deadline, we get a plan for my morning routine. A temporal network summarizes these temporal relationships, while disregarding physical state, such as what exactly happens to the dishes. (Those aspects are out of scope for scheduling, and left for a planner to consider.)

Sometimes, when we dispatch the activities of our plan, we don't have control over how long an activity lasts, but rather Nature determines it. When I walk out my apartment door and push the elevator button, residents on other floors might be calling it, too. Thus, I am at the mercy of the situation (and the opaque elevator scheduling algorithm). We could still determine an upper bound on the wait time, as there are a finite number of floors, but once I push that button, I have to accept a wait time ranging anywhere from zero (the elevator is already on my floor) to that bound. This notion of *uncontrollable duration* is the key

aspect[1] of STNUs.

In the remainder of this section, I will first focus on how the STNU model captures the semantics of uncontrollable durations in temporal plans. Once the STNU semantics are laid out, I will then define the problem of scheduling STNUs, which will require the supporting concepts of execution history and scheduling policy.

2.1.1 Modeling STNUs

To illustrate, I will model as an STNU a situation where you make spaghetti for your family. I do this in two parts: first just the activities, and then with requirements imposed.

Example 2.1 (Family spaghetti: activities)**.** *It's Saturday noon, and Chef You are going to make spaghetti and meatballs lunch for your family. Figure 2-1 delineates your plan[2] in graphical form. There are three ingredients that need to be cooked: meatballs, spaghetti, and marinara sauce. You can put the meatballs in the oven, while in parallel you boil the spaghetti and heat the sauce on two burners. Once the spaghetti is cooked* al dente*, it's simply a matter of straining it, tossing it in the sauce, and topping it with the meatballs and cheese.*

Each activity has its own flexible time window, and if you were making spaghetti just for yourself, you would be able to control the duration of each within its window. But since your family's job is to distract you, you might get lost in conversation and nearly forget to strain the spaghetti before it turns to mush. Fortunately, you've set a timer that forces you to take it off the burner once 11 minutes has passed. (It also beeps if you attempt to undercook the spaghetti by taking it off before the 8-minute mark.) You have similar timers for the other cooking activities, but not for the remaining ones, whose durations are so short that you are focused and thus at no risk of under- or over-shooting.

[1]The other aspect is due to the word *simple*, which means durations' domains must be unbroken intervals on the real numbers. Once we start saying a duration could lie within disjoint time windows, our model becomes disjunctive. This starts bringing in combinatorial complexity, which is typically dealt with in planning and not scheduling.

[2]Technically, it specifies just the STNU *temporal structure* of your plan. The plan itself would contain additional state information, such as preconditions and effects for the activities. For rhetorical convenience, I may use the term "plan" throughout to refer to such temporal structure.

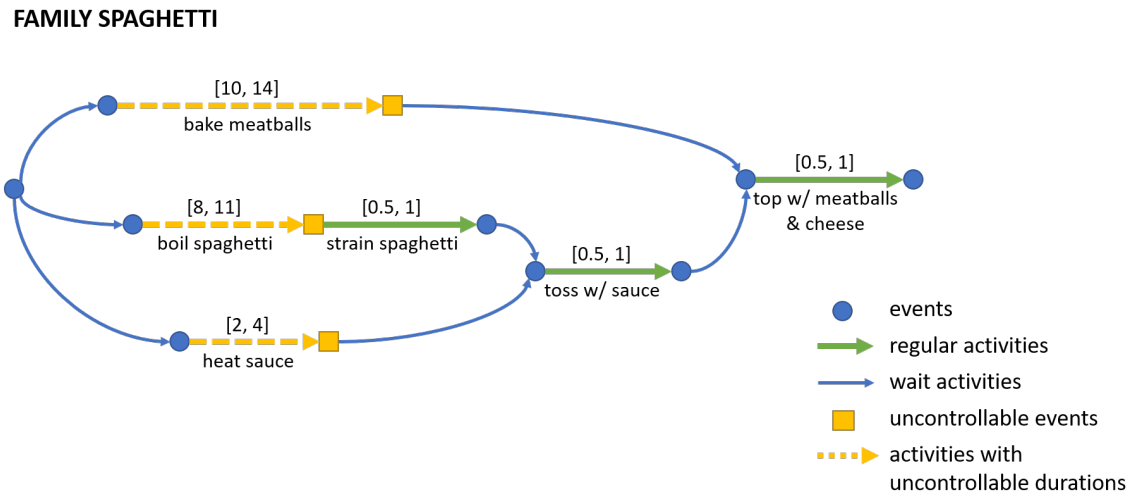

Figure 2-1: To make spaghetti for your family, these are the activities comprising your plan, labeled with time windows. This model captures the distinction between controllable and uncontrollable activities, and the waits linking them together. Together, these elements provide a roadmap for your plan's execution.

As Figure 2-1 shows, you begin by forking out three parallel threads for the three cooking activities. Once the spaghetti is strained, you merge its thread with the sauce's, so you can toss, and then merge that thread with the meatballs' for the final topping. I indicate activities as arrows, showing that they move forwards in time, and each is labeled with its flexible time window. Activities begin and end on *events*, which represent instantaneous timepoints. At any point during the plan's execution, events in the past are assigned, while events in the future are not. For instance, if you're in the middle of cooking the spaghetti, then its start event has been assigned some value t_s. However, its end event is still unassigned, though it must eventually end up in the window $[t_s + 8, t_s + 11]$.

In this example, the three cooking activities in dashed yellow have uncontrollable duration, while the remaining three activities in green are considered "regular" activities with controllable duration. The difference is that when you launch a controllable activity[3] (i.e., by dispatching its start event at a certain time), you have full control over when to dispatch its end event (as long as you stay within the bounds of that activity's flexible time window). In contrast, for uncontrollable activities, due to factors out of your control, you wait for the "arrival" of their end events, as selected by Nature. You have no knowledge of those arrivals until exactly when they happen, and all you're guaranteed is that Nature will

respect the activities' time windows. I distinguish such uncontrollable end events as yellow squares instead of blue circles.

Finally, note that the figure visually distinguishes a third type of activity: *waits*. The wait activities have implicit $[0, +\infty)$ durations, which are controllable, just like the regular activities. I don't distinguish them in Definition 2.3 below, but I do in the example because wait activities give us the semantics for merging threads:[4,5] When two or more threads merge onto a single event, they have to wait for each other to finish whatever prior activities they were engaged in.

Now so far, the example STNU doesn't contain any scheduling requirements. I've only laid out the activities and how they structurally link together, but I haven't said anything yet about any deadlines, for instance. I also haven't specified how tightly coordinated your parallel cooking threads need to be; as is, the $[0, +\infty)$ waits mean that threads would be happy to idle forever before merging.

In other words, I've only provided the execution model, which *by itself* will never experience a scheduling failure. Only by specifying scheduling requirements do I create a condition or constraint that has the possibility of being violated by and during execution. Therefore, I will now complete the example (and the STNU) by stating timing requirements on your plan.

Example 2.2 (Family spaghetti: requirements)**.** *Figure 2-2 augments the previous STNU by adding scheduling requirements in red. Your first condition is that your family is so excited to experience Chef You's World-Famous Spaghetti Lunch, that they don't want to wait more than 15 minutes. Also, the talented chef that you are, you understand the importance of*

[3]Again, for rhetorical convenience, I may use the terms "activity" and "duration" interchangeably, and both terms may be prefaced by "controllable" or "uncontrollable" as needed. Technically, temporal networks don't really contain *activities*, which belong to the plan, but just the flexible time windows that model the activities' durations. However, I want to emphasize later that these durations are distinct from *requirements*, which will look syntactically similar but are semantically different. Thus, I actually use the term "activity" in my STNU definition below, and will use it as a proxy when technically referring to an activity's duration.

[4]Waits are also useful for modeling temporal slack: By forking out onto wait activities, I allow you to stagger the start times of each cooking activity. Conversely, I chose not to include a wait between cooking and straining the spaghetti, lest you risk losing *al dente* perfection.

[5]I don't include *waits* and *threads* in my STNU definition, for the following reasons: a) as noted above, I consider waits a type of controllable duration activity; b) since threads could fork or merge anywhere, it would be easier to define them separately, and then infer them as needed from the temporal network structure; and c) I don't wish to diverge too much from existing literature.

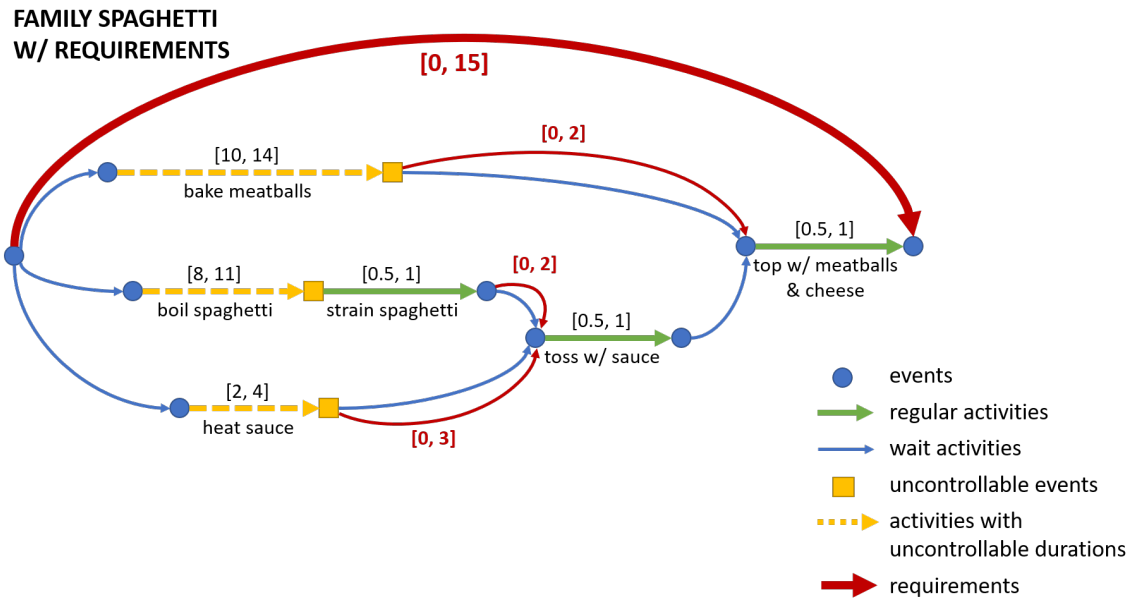

Figure 2-2: The red additions, compared to Figure 2-1, indicate timing requirements for your plan. For visual clarity in subsequent graphical depictions, I will label requirements on waits with just the red intervals, and omit the red arrows.

keeping freshly cooked ingredients steaming hot, so you have three more timing conditions[6] *within the plan:*

1. *Once the meatballs are out of the oven, you don't want them sitting out cooling for more than two minutes.*

2. *Once the spaghetti is strained, they have to be tossed in sauce within two minutes, or the strands will get starchy and stick together.*

3. *Like the meatballs, the sauce shouldn't sit around cooling for too long, but being a thick liquid, it cools slower, so you're willing to let it sit for up to three minutes.*

On the face of it, requirements look just like activities: they also span pairs of events and are labeled with flexible time windows. But instead of representing physical processes, these time windows are desired constraints on the separation between arbitrary pairs of events,

[6]These conditions effectively place restrictions on certain merging wait activities, by lowering the $+\infty$ upper bound to a finite number. In Figure 2-2, I drew these requirements on top of those waits as interval-labeled red arrows. To reduce clutter in subsequent diagrams, when there is a requirement over just a single activity, I will simply write the requirement's time window in red directly over the activity.

so there is no default "guarantee" that the plan's execution will result in those separations falling within the time windows. For example, if your family had insisted on having lunch in 10 minutes from now instead of 15, you would justifiably protest on the basis that you need at least 10 minutes to simply bake the meatballs, plus at least 30 more seconds to arrange them on top of the spaghetti.

Temporal requirements can be understood as pairs of constraints: Between any two events, they establish both a lower and upper bound on their time separation. In Example 2.2, you've actually only specified the upper bounds. That means I could have made the lower bounds $-\infty$ instead of zero, and still have captured your intent[7]. This observation leads to two points of difference between activities and requirements:

1. Requirements can have negative lower bounds, whereas activities' time windows must be strictly non-negative[8], because they represent physical processes.

2. As they specify pairs of constraints, requirements are actually reversible: A lower bound l and upper bound u in one direction are equivalent to a lower bound $-u$ and upper bound $-l$ in the reverse. For example, I could have specified the 15 minute deadline as a reverse arrow from the end of the plan to the beginning with time window $[-15, 0]$. In practice, unless a requirement is symmetric $[-x, +x]$, it will usually be more natural to write it one way over the other, but it makes no difference to the model.

Now that I've illustrated a complete STNU through this spaghetti example, I will capture its features in a formal definition of STNUs. Then I will complete the modeling discussion by noting a couple aspects of the definition that I didn't yet highlight in the example.

Definition 2.3 (STNU)**.** *A* simple temporal network with uncertainty $\mathcal{N}^u = \langle \mathcal{E}, \mathcal{E}^u, \mathcal{A}, \mathcal{A}^u, \mathcal{R} \rangle$ *is a collection of the following:*

- *Controllable events $\mathcal{E}$. Each $e_i \in \mathcal{E}$ is a real-valued variable, to be dispatched by a scheduling policy during execution.*

[7]Upper bounds of $+\infty$ and lower bounds of $-\infty$ are equivalent: they both effectively indicate that there is no constraint.

[8] I allow for the idealization of activities with zero (i.e., instantaneous) duration. In real life, you may need to account for $+\epsilon$ of time for dispatching each event, but those considerations are typically application-specific, so I don't include them in my core theory.

- *Uncontrollable events $\mathcal{E}^u$. The only difference with respect to $\mathcal{E}$ is that they are to be dispatched by Nature. Also, each $e_j^u \in \mathcal{E}^u$ serves as the end event for a unique uncontrollable activity $a_j^u \in \mathcal{A}^u$.*

- *Activities with controllable duration $\mathcal{A}$. Each $a \in \mathcal{A}$ takes the form $\langle e_i, e_j, [l, u] \rangle$, where $e_i \in \mathcal{E} \bigcup \mathcal{E}^u$, $e_j \in \mathcal{E}$, and $0 \leq l \leq u$. This specifies an activity that begins on any event e_i, ends on a controllable event e_j, and its duration $e_j - e_i$ falls in the non-negative interval $[l, u]$.*

- *Activities with uncontrollable duration $\mathcal{A}^u$. Each $a_j^u \in \mathcal{A}^u$ takes the form $\langle e_i, e_j^u, [l, u] \rangle$, and it specifies an activity that ends on a unique uncontrollable event $e_j^u \in \mathcal{E}^u$. Otherwise, it has the same conditions as a controllable activity.*

- *Requirements $\mathcal{R}$. Each $r \in \mathcal{R}$ takes the form $\langle e_i, e_j, [l, u] \rangle$, where $e_i, e_j \in \mathcal{E} \bigcup \mathcal{E}^u$ and $l \leq u$. This specifies a requirement that $e_j - e_i \in [l, u]$, regardless of whether e_i and e_j are assigned by the plan dispatcher or by Nature.*

A subtle but key property of this definition is the bijection between uncontrollable activities and uncontrollable events (and therefore $|\mathcal{A}^u| = |\mathcal{E}^u|$). Uncontrollable events arise in the first place *because* of the need to model how uncontrollable activities terminate. Namely, the end event e_j^u of an uncontrollable activity is a unique physical phenomenon, and therefore cannot exist on its own, or terminate any other activity. However, any other activity may start on e_j^u, because with respect to launching an activity, all it needs is an instantaneous start time, which any event, controllable or not, can provide. It is also the case that requirements may freely point to or away from an uncontrollable event.

To summarize, the "restrictions" on an STNU are that activities must have non-negative duration, and uncontrollable activities end on unique uncontrollable events. It may also be helpful to think of the "un-restrictions" as being that activities may start on any event, and requirements may link any pair of events with arbitrary lower and upper bounds.

When discussing the size of an STNU for complexity purposes, it turns out the two most important parameters are the number of events total and the number of uncontrollable

activities. By convention [33], I denote the former by N and the latter by K:

$$N = |\mathcal{E}| + |\mathcal{E}^u|,$$
$$K = |\mathcal{A}^u| = |\mathcal{E}^u|.$$

2.1.2 The STNU scheduling problem

Having presented the STNU model as threads of activities that must be executed in a way that satisfies scheduling requirements, I will define in this subsection what it means to perform such execution[9]. In a nutshell, I will encode the execution decisions of the user (i.e., the plan dispatcher) into a *scheduling policy*, while representing the realized durations of uncontrollable activities, determined by Nature, as *outcomes*. Together, a policy and a full set of outcomes will produce a record of how execution unfolds, known as an *execution history*, we want this history to satisfy the requirements.

To illustrate these concepts intuitively, I begin by presenting a sample execution of the family spaghetti STNU, showing how you might carry out the plan under a specific scenario. After making a few observations, I then formalize these concepts so that I can precisely state the STNU scheduling problem.

Example 2.4 (Sample execution of the family spaghetti STNU)**.** *Taking stock of your plan, you realize that baking the meatballs and boiling the spaghetti are the real timesinks, whereas heating the sauce is relatively quick, and so are the remaining activities. Therefore, you decide to waste no time putting the meatballs in the oven and the spaghetti on the burner. Now you can rest a bit, but at seven minutes in, before either the meatballs or spaghetti are done, you put the sauce on the second burner. This way, in the worst case, you can take the*

[9]The terms "execution", "dispatch", and "scheduling" are related and often seem interchangeable. In my writing, I endeavor to make the following distinctions: a) Scheduling is simply the act of deciding when activites should begin and end. It can be performed offline or online. b) Execution and dispatch both refer to the actual carrying-out of activities, which is online and in real time. Both terms can often be used interchangeably, but I use "dispatch" only to describe an event being executed or a controllable activity being initiated (via the dispatching of its start event). This is because the term connotes immediacy and something being launched. In contrast, I typically use "execution", which has a broader connotation, for describing larger-scale happenings, such as the execution of a thread, or of an entire plan. c) One exception to this rule is "plan dispatcher", which is what executes the plan by running the scheduling policy. I inherit this term from the literature, and you can think of it as execution via controlled dispatching of individual activities.

spaghetti and sauce off their burners simultaneously at $t = 11$ before you overcook them.

At this point, a relative calls you over to ask what happened to their computer and if you can fix it. Reluctantly, you start thumbing through their menus and settings, but with an eye on the desktop clock. A couple minutes in, they start asking you too many questions about what you're doing, so you cite culinary priority and glide back into the kitchen.

Now it's $t = 10$, and to look busy in the eyes of your confused relative, you decide to take the spaghetti and sauce off the heat. But then that means time is of the essence. Setting the sauce aside, you must immediately strain the spaghetti of hot boiling water, which you do carefully with raised eyebrows, telegraphing your concentration (in case your relative is watching). This take you a minute (it's a large pot for a large family). Then you spend the next minute distributing the spaghetti onto plates and mixing the sauce into each.

This brings you to $t = 12$, and your hawk-eyed relative, sensing an opening, protests at your leaving them hanging and resumes their line of questioning. Unfortunately (or fortunately), you still have the meatballs in the oven, but being the gracious tech support that you are, you summarize where they have to go look to restore their bookmarks, and then you send them away. Now it's $t = 13$, and to stave off any further interruptions, you whisk the meatballs out of the oven, and carefully arrange five on each plate in the shape of a smiley face (except for one).

At $t = 14$, you announce lunch is ready, one minute ahead of schedule. The floodgates are opened, everyone pours into the dining room, and while musical chairs is happening, you stare off into space, envisioning the rave reviews, the encouragement to open a restaurant, your first Michelin star...

But first, back to reality. You just created an execution history for your STNU, depicted in Figure 2-3. Let's review how that happened from the point of view of a scheduling policy.

While you were busy fending off complaints about Firefox's user interface, the STNU execution center of your brain was subconciously hard at work, running a scheduling policy. The first observation I want to make is that while I had described the example in terms of you performing activities, the actual *scheduling* decisions were when to start activities, and if they were controllable, when to end them. In other words, scheduling happens on the events, and not during the activities. So fundamentally, a scheduling policy decides when

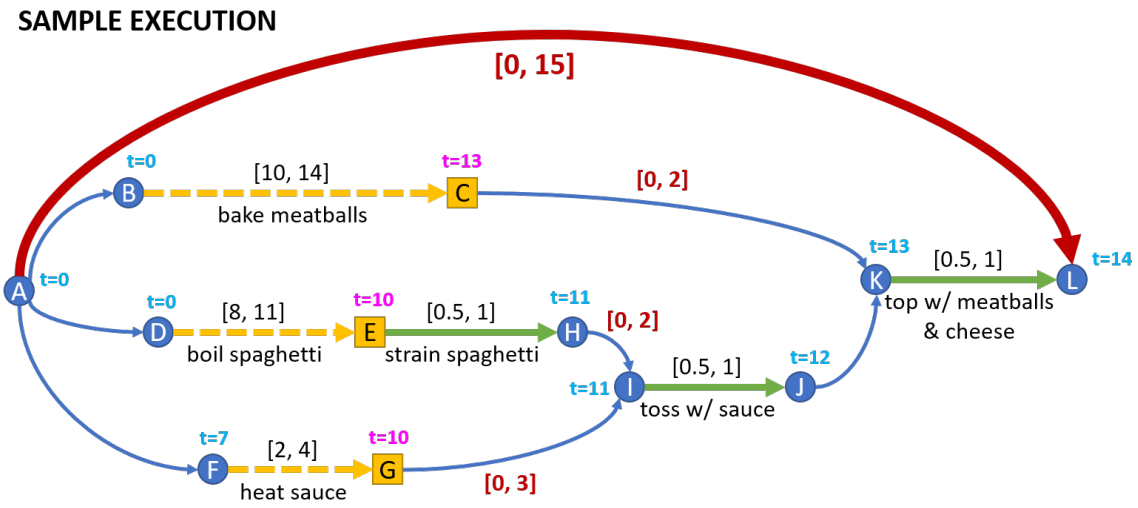

Figure 2-3: I annotate the family spaghetti STNU with times indicating when each event got dispatched, according to the sample execution in Example 2.4. Dispatchings for controllable events, as decided by the scheduling policy, are written in light blue, while those for uncontrollable events, as per Nature, are in magenta. To facilitate discussion below, I have given each event a letter name.

to dispatch events, and controllable ones at that.

The uncontrollable events, then, are dispatched by Nature. You can think of Nature's process as follows: Every time an uncontrollable activity is launched (via the dispatching of its start event), Nature selects a duration at random from the activity's time window, and decides to dispatch the activity's end event that many time units afterward. When you put the meatballs in the oven at event B, Nature already decided that you would take them out 13 minutes later at event C. You didn't know it at the time, but "Nature" summarized (and sampled) the eventual effects of dealing with your relative when it selected that duration outcome. Of course, "Nature" is a theoretical construct, and processes in real life can't see into the future (unless you want to get metaphysical). But describing it like so provides a clean semantics for how uncontrollable event dispatches happen.

The concept of an execution history can then be simply understood as a sequence of event dispatches, building up as they happen in real time. Given such a history, it's very straightforward to check whether the requirements were satisfied. In our example, you were very proactive with the tossing and topping activities, so that the waits from C to K and H to I were basically instantaneous, and the wait from G to I lasted only a minute, well

within the three-minute limit. Finally as noted, you completed the plan with one minute to spare.

Due to the uncertainty in the uncontrollable activities, running a policy multiple times may yield varying execution histories. The question then is whether all those histories will satisfy the requirements. For a negative example, suppose you had taken a minute to rest between events H and I (it's a very heavy pot). That would push J to execute no earlier than $t = 12.5$, which would still be fine under the current circumstances. But then suppose you had also decided to take out the meatballs at $t = 10$ (maybe you were in the mood for medium-well). Now you have a problem, because K can't happen earlier than J, and the difference between $C = 10$ and $K \geq 12.5$ violates the two-minute limit on how long the meatballs can sit out. To summarize, for any scenario that begins as Example 2.4 did, except the meatballs were done in 10 minutes, if your policy waits more than 30 seconds between straining the spaghetti and tossing it with sauce, you will end up with a history that leaves the meatballs sitting out for too long.

Ultimately, the STNU scheduling problem is to find a scheduling policy that is robust to any outcomes of the uncontrollable activities' durations. The preceding example and discussion a sample execution should have made this problem intuitive to understand, so at this point, I will state the formal definition upfront. Then I will build up the formal definitions of history and outcome, to support a rigorous definition of scheduling policy, which is central to the problem definition.

Problem 2.5 (STNU scheduling problem)**.** *Given an STNU $\mathcal{N}^u = \langle \mathcal{E}, \mathcal{E}^u, \mathcal{A}, \mathcal{A}^u, \mathcal{R} \rangle$, determine whether there exists a feasible scheduling policy $\mathcal{P}$, i.e., for any full outcome ω to the uncontrollable activities $\mathcal{A}^u$, repeatedly applying $\mathcal{P}$ produces a final execution history ξ that satisfies the requirements $\mathcal{R}$.*

The key word *feasible* refers to the guarantee that the final execution history will satisfy the requirements. This property is straightforward to attach to the definitions, which I do below, first for execution histories and then for policies.

The other key word I haven't touched upon is "repeatedly". I want to convey the idea that when you execute a temporal network, you incrementally build an execution history,

every time an event is dispatched. At any moment in the execution, a scheduling policy looks at the current state[10], and decides what controllable event it will try to dispatch next and when. The result of that decision is either that event will execute at the planned time, or if there are any active uncontrollable activities, one of their end events might arrive earlier than that time. Either way, the execution history would be extended by whichever event arrived earlier, thus creating a new state for the policy to operate on.

The remaining definitions in this subsection codify this intuition. First, I have to define how I represent execution histories:

Definition 2.6 (Execution history)**.** *A (full)* execution history ξ *for an STNU* $\mathcal{N}^u$ *is a sequence of monotonically increasing assignments to all the events* $\mathcal{E} \bigcup \mathcal{E}^u$*, i.e.,*

$$\xi = \big\langle (e_{i_1}, t_1), (e_{i_2}, t_2), \ldots, (e_{i_N}, t_N) \big\rangle,$$
$$\textit{where } t_1 \leq t_2 \leq \ldots \leq t_N.$$

A partial execution history $\xi_k = \big\langle (e_{i_1}, t_1), \ldots, (e_{i_k}, t_k) \big\rangle$ *only assigns* k *events, where* $0 \leq k < N$.

Thus, execution of an STNU begins with an empty partial history, and proceeds by appending successive event assignments. The full history of the sample execution in Example 2.4, and illustrated in Figure 2.4, would be:

$$\xi = \big\langle (A, 0), (B, 0), (D, 0), (F, 7), \mathbf{(\boldsymbol{E}, 10)}, \mathbf{(\boldsymbol{G}, 10)}, (H, 11),$$
$$(I, 11), (J, 12), \mathbf{(\boldsymbol{C}, 13)}, (K, 13), (L, 14) \big\rangle.$$

For clarity, I have bolded the uncontrollable events' dispatches, even though the definition of partial histories makes no distinction when they arrive. They exist to remind us that those dispatches were made by Nature, while all the others were due to scheduling decisions made by your scheduling policy.

Naturally, a history must respect activities' time windows for it to capture the execution semantics of STNUs. That is, for each activity, controllable or not, its realized duration

[10]By "state", I mean the current state of what has executed and what has not. Again, this is scheduling and not planning, so there is no "physical" state or other state spaces involved – only the "temporal" state of the events.

must fall within the activity's time window. I call such a history *valid*. In the course of an STNU's execution, invalid histories are simply never generated.

Given a valid history, the next property we desire is for it to respect the requirements $\mathcal{R}$, which I call *feasible*. Because activities and requirements share the same syntactic form, it's really the same condition drawn from an additional set of constraints. But there are a couple wrinkles when it comes to partial histories and feasibility, which I detail in the definition below.

Definition 2.7 (Valid and feasible execution histories)**.** *A full execution history ξ for an STNU $\mathcal{N}^u$ is* valid *if for all activities $a = \langle e_i, e_j, [l, u] \rangle \in \mathcal{A} \bigcup \mathcal{A}^u$, the assigned values of e_i and e_j satisfy $e_j - e_i \in [l, u]$.*[11] *Furthermore, a history is* feasible *if it is both valid* and *for all requirements $r = \langle e_i, e_j, [l, u] \rangle \in \mathcal{R}$, the assigned values of e_i and e_j satisfy $e_j - e_i \in [l, u]$.*

A partial history ξ_k is valid if the completed activities so far have respected their time windows, and any currently executing activities haven't violated theirs. So for any activity where both its start event e_i and end event e_j have been executed, i.e., are in the scope of ξ_k, the same feasibility condition above applies. And for any activity where only its start event e_i is in scope, the current time, indicated by the last assignment (e_{i_k}, t_k) in ξ_k, must not exceed the latest possible time the activity could finish, i.e., $t_k \leq e_i + u$.

Finally, a partial history ξ_k is feasible if it is valid to begin with, and then for all requirements whose events e_i and e_j are within scope, those events must respect the requirement's time window, $e_j - e_i \in [l, u]$.

The last point I wish to make about execution histories pertains to events happening simultaneously: First, I assume the policy computes instantaneously[12]. This allows it to treat simultaneous events as successive appends to the history. Second, if two events are linked by an activity with a $[0, u]$ time window, the end event has to come after the activity's start in the history. For example, $H \rightarrow I$ represents a wait activity that you spent zero time on, but it is an activity with a start and an end, so H must come first in the history. In

[11] Technically, I should be using the t values attached to e_i and e_j in the execution history. Due to the notational inconvenience of having to map subscripts i and j in an activity's events to subscripts i_k in a history, I just use e_i and e_j directly here and in the rest of the definition.

contrast, B and D could be interchanged, because they represent independent start events of different activities, even though you dispatched them simultaneously with A.

Next, to describe how uncontrollable events get added to a history, I need to state how Nature determines the durations of the activities those events terminate. When discussing Example 2.4 prior, I stated that once you dispatch an uncontrollable activity, Nature immediately, behind the scenes, chooses a duration for it at random. This was intuitive for a first explanation, but in the definition below, I modify it slightly to say that Nature samples all the uncontrollable durations prior to execution of the entire plan.

The main reason I say this is that there could be dependencies and/or correlations between activities' durations. For instance, because you were cooking the spaghetti and the sauce on parallel burners, you might be inclined to take them off together (as you did) if the current partial execution history allows for it. Thus, due to potentially complex and unmodeled interactions between activities' durations, it is more correct to say that Nature "foresees" the entire outcome before plan execution, rather than independently sampling each duration as it comes.

This is a small and digestible modification for defining outcomes, but you will shortly see that it streamlines Definition 2.10, which explains the result of a scheduling decision. It will also turn out to be helpful in the next section, when I make the durations' uncertainty probabilistic, and we'll need to assign likelihoods to outcomes.

Definition 2.8 (Outcome)**.** *Let ω_j be a random variable representing the realized duration of uncontrollable activity a^u_j. The domain Ω_j of variable ω_j is the activity's time window. Each value in Ω_j is an* outcome *for ω_j.*

The space of all outcomes for all uncontrollable activities is:

$$\Omega = \Omega_1 \times \Omega_2 \times \cdots \times \Omega_k.$$

Therefore, a full outcome *$\omega \in \Omega$ is a vector of outcomes to all the uncontrollable activities,*

[12]This relates to my earlier point in footnote 8 about allowing activities to have zero duration. In reality, online decision-making takes time, but as long as it is insignificant compared to the timescale of the activities' durations (and we do have efficient algorithms), modeling it as instantaneous is a reasonable approximation.

i.e,

$$\omega = \langle \omega_1, \omega_2, \ldots, \omega_K \rangle.$$

Just prior to the execution of an STNU $\mathcal{N}^u$, Nature selects a full outcome ω at random; this selection is known only to Nature and is not revealed to the scheduling policy. During execution, whenever an uncontrollable activity a_j^u begins, Nature will automatically dispatch the end event e_j^u exactly ω_j time units afterward, regardless of the policy's decisions.

In Example 2.4, the three uncontrollable cooking activities form a three-dimensional space of outcomes. If ω_1 is the duration for baking meatballs, ω_2 for boiling spaghetti, and ω_3 for heating sauce, then the full outcome in that example is $\omega = \langle 13, 10, 3 \rangle$.

Having laid out definitions for execution histories and outcomes, I can finally state what exactly is a scheduling policy and how it generates a full execution history. I do this in two parts: first with a straightforward definition of what a policy outputs, and then a more intricate description of how that output contributes to the execution history.

Definition 2.9 (Scheduling policy)**.** *A* scheduling policy *$\mathcal{P}$ for an STNU $\mathcal{N}^u$ is a function that maps a valid partial execution history ξ_k into a scheduling decision σ. This decision may be one of two forms: 1) a proposed assignment (e_{i_l}, t_l), which signals the intent to dispatch some unexecuted* controllable *event $e_{i_l} \notin \texttt{Scope}(\xi_k)$ at time $t_l \geq t_k$, which could be now or in the future; or 2) a* `wait` *decision*[13]*.*

A policy may output a `wait` *only if ξ_k indicates some uncontrollable activity is currently in progress. I.e., there must exist $a_j^u = \langle e_i, e_j^u, [l, u] \rangle \in \mathcal{A}^u$ such that $e_i \in \texttt{Scope}(\xi_k)$ and $e_j^u \notin \texttt{Scope}(\xi_k)$. (And as a sanity check, if ξ_k is valid, then $e_i + u \geq t_k$.)*

I have two points to make about this definition. First, as you can see, a policy does not directly effect the execution history, but rather signals its intent to do so via a decision. The following definition presents the result function $\mathcal{O}$, which then transforms that intent into a realized extension of the history.

[13]This `wait` should not be confused with the wait activities in the STNU model. As I mentioned earlier, wait activities are considered a type of activity with controllable duration, so they are intrinsic to the STNU form, which is ultimately the object being executed. In this definition, I am talking about the actions that the *policy* may take in order to perform said execution on an STNU. So a `wait` action exists outside of an STNU, and Definition 2.10 specifies its effect on that STNU's execution.

Second, the purpose of `wait` decisions is to allow policies to wait for uncontrollable activities to conclude before proceeding. This is necessary, for instance, if there are only uncontrollable activities towards the end of a plan. Once those activities begin, there will be no more controllable events for the policy to schedule, so it can only `wait` for the final outcomes. It wouldn't make sense, however, to `wait` in the opposite situation, when there remain only controllable events to dispatch, because execution would simply hang. Hence, I place the restriction that `wait`s may occur only if there is currently some uncontrollable event to wait on.

Note that it is entirely possible for a policy to use a `wait` in the middle of a plan, when there are still controllable events left. Suppose a thread contains an uncontrollable activity $A \dashrightarrow B$ followed directly by a controllable one $B \rightarrow C$. Thus, B is an uncontrollable event, and C is controllable. Given an execution history where A has been executed but B has not, the policy could eagerly try to schedule C in the future. But B's arrival would preempt it anyway, forcing the policy to try again. So an alternative solution would be to just `wait` (or try to scheduling something else on some other thread), allow B to arrive, and then decide on a time for C.

Because execution histories must always grow monotonically in time, the key to deciding the result of a policy decision is to determine the earliest next arrival of an uncontrollable event, if any. And if that arrival is earlier than when we had proposed to dispatch a controllable event, then our proposal gets *preempted* by that arrival. This next definition explains the exact mechanism of preemption, and how it factors into determining the result.

Definition 2.10 (Result of a policy decision)**.** *Given a partial execution history ξ_k, and a decision $\sigma = \mathcal{P}(\xi_k)$ generated by some policy $\mathcal{P}$, the* result *function $\mathcal{O}$ determines the result (or outcome[14]) of applying σ to ξ_k. The result function is parameterized on a full outcome ω to the uncontrollable activities, which was preselected by Nature before execution began. The result $\mathcal{O}(\xi_k, \sigma; \omega)$ is always a new execution history ξ_{k+1} that extends ξ_k by an event dispatch $(e_{i_{k+1}}, t_{k+1})$:*

$$\xi_{k+1} = \big\langle (e_{i_1}, t_1), \ldots, (e_{i_k}, t_k), (e_{i_{k+1}}, t_{k+1}) \big\rangle.$$

This new history is a partial one as well, unless $k = N - 1$, in which case ξ_{k+1} is the full and final history.

If σ is of the form (e_{i_l}, t_l), then either e_{i_l} gets dispatched as planned, and so $\mathcal{O}$ simply appends σ to ξ_k; or, some other uncontrollable event e_j^u preempts σ by arriving earlier (or simultaneously) at time $t_j \leq t_l$, and $\mathcal{O}$ appends (e_j^u, t_j) to ξ_k. The way $\mathcal{O}$ decides is as follows:

1. *If there are no uncontrollable activities in progress, then $\mathcal{O}$ dispatches event e_{i_l} according to σ.*

2. *Otherwise, $\mathcal{O}$ uses the full outcome ω to determine the end times of the activities in progress. For each of those activities $a_j^u = \langle e_i, e_j^u, [l, u] \rangle$, $\mathcal{O}$ computes that Nature will dispatch e_j^u at time $t_j = t_i + \omega_j$, where t_i is when ξ_k says e_i got dispatched, and ω_j is the predetermined outcome for a_j^u's duration.*

3. *Finally $\mathcal{O}$ finds the minimum $t_{j,min}$ among all the t_j and compares it to t_l. If $t_l < t_{j,min}$, then $\mathcal{O}$ dispatches e_{i_l} just like in step 1. Otherwise, $\mathcal{O}$ finds the uncontrollable event $e_{j,min}^u$ that Nature dispatches at $t_{j,min}$ and appends $(e_{j,min}^u, t_{j,min})$ to ξ_k. If there are multiple such events, $\mathcal{O}$ selects one at random. (The other(s) will get their turn in subsequent calls to $\mathcal{O}$.)*

The other possibility for σ is that it's a `wait`*, which the policy may generate only if uncontrollable activities are in progress. In this case, $\mathcal{O}$ uses the same logic as in steps 2 and 3 above, minus the reasonining about t_l, to find the uncontrollable event $e_{j,min}^u$ that arrives first.*

Together, my definitions for $\mathcal{P}$ and $\mathcal{O}$ imply the scheduling mechanism behind plan execution: To begin with, Nature chooses behind our backs a full outcome ω to all the uncontrollable events. Then we as the plan dispatcher step in, and starting from the empty

[14]Luke Hunsberger [33] uses the term "outcome" and the notation $\mathcal{O}$ for the function that produces it. Since I already used "outcome" to describe what Nature selects for the uncontrollable durations, I use the term "result" to indicate the consequence of and policy's decision. But since I already use $\mathcal{R}$ for requirements, and due to my reluctance to wade into capital Greek letters, I preserve Hunsberger's $\mathcal{O}$ notation. I think it also nicely complements $\mathcal{P}$ for the policy function. Hunsberger had used "situation" to refer to a realization of the uncontrollable durations, but I feel "outcome" is more specific and applicable, because it connotes sampling from a stochastic process.

execution history $\xi_0 = \langle\rangle$, we apply $\mathcal{P}$ to it to obtain the first decision $\sigma_1 = (e_{i_1}, t_1)$. Typically, e_{i_1} will be the start of some thread, and t_1 will be 0. Next we apply $\mathcal{O}$ to ξ_0 and σ_1, conditioned on ω, and $\mathcal{O}$ sees that there are no uncontrollable activities in progress yet. So it simply appends σ_1 to ξ_0, thus creating $\xi_1 = \langle(e_{i_1}, t_1)\rangle$.

Subsequently, the plan dispatcher applies $\mathcal{P}$ to generate σ_2, but the difference now is that some activity is in progress. Thus, the preemption logic in $\mathcal{O}$ comes into full force when determining ξ_2. This goes on, with $\mathcal{P}$ and $\mathcal{O}$ alternatingly operating on each other's outputs, until we get a full execution history $\xi = \xi_N$. Because there are N events, and each application of $\mathcal{O}$ appends exactly one event dispatch onto the history, there must be exactly N rounds of history generation.

For a fully grounded example, I return to the sample execution of the family spaghetti scenario in Example 2.4, and I present a trace of $\mathcal{P}$ and $\mathcal{O}$ operating on each other to produce the final history.

Example 2.11 (Sample execution in terms of $\mathcal{P}$ and $\mathcal{O}$)**.** *The sequence of $N = 12$ steps, plus the initial state, is as follows:*

0. *Prior to execution, Nature selects the durations for the uncontrollable activities, and you have an empty execution history.*

$$\omega = \langle 13, 10, 3\rangle$$
$$\xi_0 = \langle\rangle$$

1. *The first decision you make (as the policy $\mathcal{P}$) is to launch the start event A at $t = 0$.*

$$\sigma_1 = \mathcal{P}(\xi_0) = (A, 0)$$
$$\xi_1 = \mathcal{O}(\xi_0, \sigma_1; \omega) = \langle(A, 0)\rangle$$

2. *You also decide to immediately begin baking the meatballs and boiling the spaghetti.*

This happens in two steps, both instantaneously. First you dispatch B at $t = 0$.

$$\sigma_2 = \mathcal{P}(\xi_1) = (B, 0)$$
$$\xi_2 = \mathcal{O}(\xi_1, \sigma_2; \omega) = \big\langle (A, 0), (B, 0) \big\rangle$$

3. *Then D at $t = 0$.*

$$\sigma_3 = \mathcal{P}(\xi_2) = (D, 0)$$
$$\xi_3 = \mathcal{O}(\xi_2, \sigma_3; \omega) = \big\langle (A, 0), (B, 0), (D, 0) \big\rangle$$

4. *Now you have uncontrollable events C and E on the horizon. Nature's already decided to dispatch them at $t = 13$ and $t = 10$ respectively, but you don't know that. You just know that they can't arrive earlier than $t = 10$ and $t = 8$, respectively. So you choose to begin heating the sauce at $t = 7$ by dispatching F, and $\mathcal{O}$ obliges.*

$$\sigma_4 = \mathcal{P}(\xi_3) = (F, 7)$$
$$\begin{aligned} \xi_4 &= \mathcal{O}(\xi_3, \sigma_4; \omega) \\ &= \big\langle (A, 0), (B, 0), (D, 0), (F, 7) \big\rangle \end{aligned}$$

5. *Now you have a third uncontrollable event G on the horizon, and all three are blocking subsequent controllable events, so all you can do is wait. At $t = 10$, Nature dispatches E and G. Because $\mathcal{O}$ can only append one event at a time, it selects E.*

$$\sigma_5 = \mathcal{P}(\xi_4) = \texttt{wait}$$
$$\begin{aligned} \xi_5 &= \mathcal{O}(\xi_4, \sigma_5; \omega) \\ &= \big\langle (A, 0), (B, 0), (D, 0), (F, 7), \boldsymbol{(E, 10)} \big\rangle \end{aligned}$$

6. *Once E got dispatched, that launched the controllable activity to strain the spaghetti, and you decide that will take one minute, so $\mathcal{P}$ schedules H for $t = 11$. Unbeknowst to you (at this point within the processing of $t = 10$), G has also arrived, and that*

preempts your decision for H.

$$\begin{aligned}\sigma_6 &= \mathcal{P}(\xi_5) = (H, 11)\\ \xi_6 &= \mathcal{O}(\xi_5, \sigma_6; \omega)\\ &= \big\langle (A,0), (B,0), (D,0), (F,7), \boldsymbol{(E,10)}, \boldsymbol{(G,10)} \big\rangle\end{aligned}$$

7. *So you try again, and this time H succeeds.*

$$\begin{aligned}\sigma_7 &= \mathcal{P}(\xi_6) = (H, 11)\\ \xi_7 &= \mathcal{O}(\xi_6, \sigma_7; \omega)\\ &= \big\langle (A,0), (B,0), (D,0), (F,7), \boldsymbol{(E,10)}, \boldsymbol{(G,10)}, (H,11) \big\rangle\end{aligned}$$

8. *Immediately after straining, you merge the spaghetti thread with the sauce's, and commence tossing them together by requesting event I to be dispatched right now $t = 11$. Remember that by this point, we are in the time window for C's potential arrival, so event I could be preempted. But $\mathcal{O}$ looks at ω and determines event I gets executed as planned.*

$$\begin{aligned}\sigma_8 &= \mathcal{P}(\xi_7) = (I, 11)\\ \xi_8 &= \mathcal{O}(\xi_7, \sigma_8; \omega)\\ &= \big\langle (A,0), (B,0), (D,0), (F,7), \boldsymbol{(E,10)}, \boldsymbol{(G,10)}, (H,11)\\ &\quad (I,11) \big\rangle\end{aligned}$$

9. *You follow up event I's successful dispatch with J at $t = 12$, still not preempted by*

C.

$$\begin{aligned}
\sigma_9 &= \mathcal{P}(\xi_8) = (J, 12) \\
\xi_9 &= \mathcal{O}(\xi_8, \sigma_9; \omega) \\
&= \big\langle (A,0), (B,0), (D,0), (F,7), \boldsymbol{(E,10)}, \boldsymbol{(G,10)}, (H,11) \\
&\qquad (I,11), (J,12) \big\rangle
\end{aligned}$$

10. *Finally, you are just waiting for the meatballs to pop out of the oven. At $t = 12$, you know that you can't dispatch K until C arrives, which could happen as late as $t = 14$. You also calculate that to meet the 15 minute deadline, you can afford to dispatch K as late as $t = 14.5$, At this point, you could just* `wait`*, but let's say you eagerly schedule K for $t = 14$. It turns out that gets preempted by C's arrival at $t = 13$.*

$$\begin{aligned}
\sigma_{10} &= \mathcal{P}(\xi_9) = (K, 14) \\
\xi_{10} &= \mathcal{O}(\xi_9, \sigma_{10}; \omega) \\
&= \big\langle (A,0), (B,0), (D,0), (F,7), \boldsymbol{(E,10)}, \boldsymbol{(G,10)}, (H,11) \\
&\qquad (I,11), (J,12), \boldsymbol{(C,13)} \big\rangle
\end{aligned}$$

11. *Now you just want to get the dish out as soon as possible, because all this reasoning about $\mathcal{P}$'s and $\mathcal{O}$'s is making you dizzy. So you revise your previous decision, and you tell $\mathcal{O}$ you want to dispatch K now at $t = 13$. Since there are no more uncontrollable events left, $\mathcal{O}$ happily obliges—*

$$\begin{aligned}
\sigma_{11} &= \mathcal{P}(\xi_{10}) = (K, 13) \\
\xi_{11} &= \mathcal{O}(\xi_{10}, \sigma_{11}; \omega) \\
&= \big\langle (A,0), (B,0), (D,0), (F,7), \boldsymbol{(E,10)}, \boldsymbol{(G,10)}, (H,11) \\
&\qquad (I,11), (J,12), \boldsymbol{(C,13)}, (K,13) \big\rangle
\end{aligned}$$

12. —and again when you request L to finish the plan at $t = 14$.

$$\begin{aligned}
\sigma_{12} &= \mathcal{P}(\xi_{11}) = (L, 14) \\
\xi_{12} &= \mathcal{O}(\xi_{11}, \sigma_{12}; \omega) \\
&= \big\langle (A,0), (B,0), (D,0), (F,7), \boldsymbol{(E,10)}, \boldsymbol{(G,10)}, (H,11) \\
&\qquad (I,11), (J,12), \boldsymbol{(C,13)}, (K,13), (L,14) \big\rangle
\end{aligned}$$

This example, though lengthy, illustrates a couple key features of how the result function $\mathcal{O}$ operates. First, we saw two instances of uncontrollable preemptions: G preempted H at $t = 10$, and C preempted K at $t = 13$. In the first instance, you were unfazed about H and just tried again, but in the second, you chose to bump up K from $t = 14$ to $t = 13$. Thus, preemptions give you (the policy $\mathcal{P}$) the opportunity to change your mind as a form of dynamic reactivity.

Second, we saw several instances of simultaneous events: A, B, and D all fell at $t = 0$; E and G were at $t = 10$, and C and K were at $t = 13$. These illustrate the earlier point I made, when defining execution histories, that simultaneous events are appended through successive applications of $\mathcal{P}$ followed by $\mathcal{O}$. Simultaneous events might be rare in practice (unless you're working with coarse-grained discrete times), but we still have to handle them in our definition. It would have been possible to lump them together into a single "super-assignment" that gets appended onto the history (Hunsberger [33] does this), and a realistic implementation might handle them this way. But I decided to frame them as multiple rounds to reduce the machinery needed in the definitions.

One consequence of this framing is that uncontrollable events always appear before controllable ones at the same time (see step 3 in Definition 2.10), with one exception. For example, if you had initially scheduled K for $t = 13$ instead of $t = 14$, event C's arrival at $t = 13$ would still preempt it. This is convenient, because it gives you a chance to reschedule K if you wish (e.g., at $t = 14$). But it is also necessary, because conceptually, K terminates a controllable activity (a wait) that begins on C; therefore C should precede K in the execution history. This reasoning also points to the exception, which is if you have an uncontrollable activity $U \dashrightarrow V$ with an outcome of zero duration, then U will

appear before V. That is because Nature won't consider $U \dashrightarrow V$ to be in progress until U is dispatched, and only then, on the next round will Nature preempt whatever decision you make with V's immediate arrival.

Lastly, note that in the example, I never actually specified what $\mathcal{P}$ *is*, just what decisions it outputs. Fully specifying an arbitrary policy $\mathcal{P}$ would be intractable, as I'd have to map over the entire space of execution histories, which in turn, accounts for the entire space of outcomes. However, the concept of such a mapping is well-defined, and it would be readily supplied by an algorithm. Morris [38] [39] has sketched such an algorithm for $\mathcal{P}$ when a feasible policy (see Definition 2.13) exists, and Hunsberger [33] explicitly provides one and analyzes its properties.

The following definition summarizes the entire process demonstrated in Example 2.11.

Definition 2.12 (Execution semantics)**.** *Given a scheduling policy $\mathcal{P}$ for an STNU $\mathcal{N}^u$, execution proceeds as follows:*

1. *Nature selects a full outcome $\omega \in \Omega$ at random, while the plan dispatcher initializes an empty execution history ξ_0.*

2. *At each step from $i = 1$ to $i = N$, the plan dispatcher generates a new decision $\sigma_i = \mathcal{P}(\xi_{i-1})$. Then the result function responds by extending the previous execution history to $\xi_i = \mathcal{O}(\xi_{i-1}, \sigma_i; \omega)$.*

The final execution history ξ_N is a deterministic product of the dispatcher's policy $\mathcal{P}$ and the full outcome ω referenced by $\mathcal{O}$. Therefore, I refer to it as $\xi_N = \xi(\mathcal{P}, \omega)$.

Recall that the goal of the STNU scheduling problem in Problem 2.5 is to find a *feasible* scheduling policy. With the above definition of execution semantics giving us a well-defined final execution history, it becomes quite simple to state what feasibility means for a policy, thus completing the scheduling problem definition.

Definition 2.13 (Feasible scheduling policy)**.** *A scheduling policy $\mathcal{P}$ is* feasible *for an STNU $\mathcal{N}^u$ if for every possible full outcome $\omega \in \Omega$, the final execution history $\xi(\mathcal{P}, \omega)$ is feasible.*

At this point, I have fully presented all the necessary concepts to support my statement of the STNU scheduling problem. I conclude this subsection by making one last major point

about policies, which is to distinguish between static and dynamic variants. As presented, my definition of scheduling policy has full access to the current partial history. That means when making its next decision, the policy may observe and react to uncontrollable outcomes, as well as its own prior decisions. In other words, my definition *by default* specifies a dynamic policy.

In Example 2.4, I had illustrated dynamic decision-making at several points. For example, consider what happened right after you wrapped your fascinating discussion on web browsers in order to attend to the nearly overdone meatballs. At that moment $t = 13$, you observed the arrival of event C. Then knowing that J had already executed at $t = 12$ so that the two threads were ready to merge onto K, you immediately dispatched K at the same time $t = 13$.

Sometimes, though, you may wish to not have to make scheduling choices online. Suppose word of your culinary talent got out, but in the height of the pandemic, restaurants are hard to open, so you settle for publishing a cookbook. You present your readers this wonderful STNU, but to reduce their cognitive load when they execute the plan, you don't want them to have to calculate on-the-fly when to dispatch each controllable event. Could you work out a schedule *a priori* so that no matter what distractions they might face in their own cooking activities, they can dispatch all the controllable events at set times according to you?

Unfortunately, you quickly see that's not possible. Say like you did, you ask them to start baking the meatballs at $t = 0$. In the longest case, the meatballs don't come out until $t = 14$, so to be safe, you preassign $K = 14$. But wait, other readers who are eager beavers might take the meatballs out at $t = 10$, and now they have to wait four minutes before adding them to the spaghetti and sauce. By then, the meatballs will have lost their steaming quality, and those readers will be biting into stale crusts. Either you dash your hopes for a "Cooking is Easy!" bestseller, or you find another recipe to write up, and file this one under "Family Secrets".

Silly example aside, the difference in a nutshell is that static policies must preassign all the controllable events[15], and follow that template during online execution, whereas dynamic policies have the flexibility to modify their decisions based on different realized

outcomes. As I noted above, I originally defined policies in the dynamic sense, so for static policies, I just impose restrictions on the original definition, codified below.

Definition 2.14 (Static scheduling policy)**.** *A* static *scheduling policy $\mathcal{P}$ is associated with a predetermined, full execution history $\xi^{\setminus u}$ for just the controllable events. That is, for all $e_i \in \mathcal{E}$, this predetermined history takes the form:*

$$\xi^{\setminus u} = \big\langle (e_{i_1}, t_1), (e_{i_2}, t_2), \ldots, (e_{i_{N-K}}, t_{N-K}) \big\rangle,$$
$$\textit{where } t_1 \leq t_2 \leq \ldots \leq t_{N-K}.$$

When given a partial history ξ_k, $\mathcal{P}$ finds the latest event in $\xi^{\setminus u}$ that has been executed, i.e., it identifies the largest l such that $e_{i_l} \in \texttt{Scope}(\xi_k)$. As long as $l < N + K$, there remains further controllable events to dispatch, so $\mathcal{P}$ output the decision $(e_{i_{l+1}}, t_{l+1})$, thus proposing to dispatch $e_{i_{l+1}}$, the next event in $\xi^{\setminus u}$, at its planned time. Otherwise, $\mathcal{P}$ outputs `wait`.

Note that the distinction applies only to the policy definition. For both static and dynamic policies, the execution effects of their decisions are subject to the same set of rules laid out in Definition 2.10. One way to interpret the execution of a static policy is that if one of its decisions gets preempted by an uncontrollable event arriving earlier, the policy will simply try again. Static policies stubbornly insist that their intended event get added to the execution history at the intended time, before moving on to the next. Dynamic policies don't have this restriction; if preempted, they can modify the intended time, or even switch to a different event altogether.

[15] Another seemingly reasonable interpretation of "static policy" would be that you decide on the *duration* of each controllable activity beforehand. However, this definition would pose problems for merging threads. For example, consider when the spaghetti and the sauce threads merge into I. Because of the uncontrollable $D \to E$ and $F \to G$ activities, there's no guarantee (and very unlikely) that rigidly set durations for the remaining controllable activities would result in the two threads meeting up perfectly at I.

One way to fix this might be to not preassign wait activities' durations. But you'd still be left with the question of how long to wait, and if you don't decide offline, then you'd need a strategy for deciding online, which turns it into a dynamic decision. Therefore, it's semantically cleaner to define static policies as making static decisions about the controllable events, rather than activity durations.

It turns out this definition does imply a small degree of dynamic execution: Any controllable activities that begin off of an uncontrollable event will have their end events preset, but their start events variable. Thus, the plan dispatcher would effectively have to "dynamically" decide such activities' durations. But this is a very local calculation in that you don't have to consider information from other portions of the plan, e.g., other threads or even other activities on your thread. So I don't consider this calculation to violate the spirit of a static policy.

Defining static policies as a restricted form of dynamic policies makes it clear that the former are technically a subset of the latter. So if we are looking for feasible static policies to an STNU, then we have shrunk the solution space. However, the nature of the restriction also means that it is easier to build algorithms that find static policies, and verify them (i.e., check they are actually feasible).

Indeed, the standard algorithm for determining whether an STNU admits a static policy reduces to calculating single-source shortest paths (SSSP) [60], which runs in $O(NM)$ time using Bellman-Ford. Constructing such a policy then typically requires the all-pairs shortest paths matrix (APSP), which takes $O(N^2 \log N + NM)$ time using Johnson's algorithm. Both of these are faster than the $O(N^3)$ state-of-the-art for finding a dynamic policy [39].

This concludes my exposition of the STNU scheduling problem, as specified in Problem 2.5. And based on the discussion just now, there are actually two variants of the problem: we can either look for feasible static policies, which are limited in their existence and abilities, or we can seek feasible dynamic policies, which are more powerful but likely require more complex algorithms to extract. If an STNU $\mathcal{N}^u$ has a feasible static policy, then $\mathcal{N}^u$ is called ***strongly controllable***. Analogously, if $\mathcal{N}^u$ admits a feasible dynamic policy, then it is said to be ***dynamically controllable***.

All the concepts above will carry over when I present my probabilistic version of temporal networks. Therefore, for my definitions in the next section, I will focus on the extensions and any modifications that they imply to the ones above.

2.2 The pSTN model and the chance-constrained scheduling problem

Recall the morning routine scenario I outlined at the beginning of Section 2.1. When it came to calling the elevator, I argued that there must be some finite upper bound on how long I would have to wait for it, but I never actually calculated what that might be. It turns out the numbers are not so merciful. In the worst case, the elevator would have to stop at every floor on the way up, then skip my floor, continue stopping every floor until it reached

the top, and then come down stopping every floor as well. If I assume the doors open and close perfectly each time (which they don't) and that residents shuffle on and off the elevator expeditiously without holding the door for their roommate running down the hall (which they do), we're looking at ~20 seconds between floors. For any moderately sized apartment building[16], that's a worst case of several minutes of waiting for the elevator, during which I could have wolfed down a second breakfast.

I am happy to say that I rarely encounter such comically bad luck (with the elevators). (And if I ever had, I probably just took the stairs before waiting for the final outcome.) That means it would be just as comically unsuitable to model my elevator waiting duration as an interval-bounded time window from 0 up to that bound. Therefore, the STNU model may not be entirely appropriate or sufficient to capture certain temporal plans. For the sake of shoehorning your plan into an STNU format (so that you might take advantage of efficient, well-established STNU algorithms), you might be willing to compromise model fidelity, such as by guessing a more reasonable upper bound. But then, you may have no idea how good your guess is, i.e., how likely it is that your guess is right. And should execution prove your guess to be wrong, you'd have no strategy to continue, because STNU policies technically aren't defined for outcomes that lie outside the time windows. (Oh, no! I can't make it to a lab meeting on time because the elevator didn't come in 45 seconds like I expected it to. Should I just give up on the plan and go back to bed?)

The information that STNUs' interval bounds don't capture is the *likelihood* of any given outcome. In my experience, most days I wait less than a minute for the elevator to arrive. That means from a scheduling viewpoint, I shouldn't spend much effort accounting for the possibility of waiting two, three, even four minutes, because those are precious minutes I could spend on extra sleep or proper teeth-brushing. More generally (and seriously), modeling the likelihoods of uncontrollable durations gives us justification for reducing the range of uncertainty that our policies have to consider. This in turn allows us to schedule

[16]One might argue that multiple elevators are commonplace in such buildings, and they should reduce the waiting time by a significant factor. Still, I could devise pathological counterarguments to preserve the worst case: Maybe all the other elevators are either undergoing maintenance or being held hostage by a resident in the lobby who's chatting with someone going out. Or, given the opacity of manufacturers' elevator dispatching algorithms, a sequence of perfectly timed calls across all floors may result in all elevators in staggered formation, or all but one cycling between adjacent floor pairs.

and plan less conservatively, either by giving the scheduling policy more flexibility in its decisions, or by fitting in more activities. In short, it widens the space of plans that admit feasible scheduling policies.

Fortunately, with our understanding of the STNU model and all its associated concepts, we already have most of the infrastructure needed to support duration likelihoods. All I need to do is turn each uncontrollable activity's duration from an interval-bounded range of uncertainty into a probability distribution. Specifically, in order to respect the semantics of activity duration, the distributions would be defined over non-negative time, relative to their activities' start events. In a sense, we would be stretching the domains of all the uncontrollable durations to $[0, +\infty)$, and then mapping probability density functions over those domains.

How would we obtain such distributions? If an activity could be modeled as some well-defined stochastic process, there might be an analytic form for the distribution of its duration. For example, introductory probability classes often "demonstrate" that you can model the time you spend waiting at a bus stop as an exponential distribution. However, anyone who has ever waited for a bus knows that the underlying assumptions in that argument don't match reality, which is a complex interaction of bus schedules, traffic conditions, and driver behavior. So, analytic distributions may be better suited for plans involving particle physics.

Another approach would be to gather enough samples of the duration in isolation, such that you effectively build a histogram that summarizes all the hidden complexity. This would be appropriate for my elevator scenario, as I could run a stopwatch every morning for three months to collect ~100 samples. Or, I could parallelize it all by sending a survey to the residents, and any visitors to the building would observe the curious sight of residents in each elevator lobby staring at stopwatches on their phones. Once I have my histogram, it's quite simple to perform (one-dimensional) integrals, and to estimate derivatives on it as needed.

Sometimes, exact samples are not so easy to obtain. If you're dealing with an expensive underwater robot that's time-consuming to deploy, you would be justifiably averse to dropping it in the ocean 100 times to estimate how quickly its (likely underpowered) propulsion

system can fight the effects of current. In these cases, proxy deployments (e.g., extrapolate data from safe harbor exercises) or computer simulations would be reasonable alternatives.

Lastly, most distributions for activities' durations turn out to be unimodal. (If not, then there is most likely an uncontrollable choice being made between two different processes, and this should be addressed in the planning stage.) Therefore, if you really wanted an analytic form, you could approximate your histogram via a normal distribution (or any other unimodal curve of your choice), and then truncate it below zero and renormalize.

To see probabilistic durations in context, I revisit the STNU spaghetti example, and turn it into a *probabilistic STN*.

Example 2.15 (Restaurant spaghetti)**.** *Word of your World-Famous*[17] *Spaghetti Lunch has finally spread, and when it reached the local billionaire, they wrote you a giant check to let you start your own restaurant. Now you spend your days in the back office, feet on the desk, dreaming of billions served, while a small army of apprentice chefs outside your window churns out heaps of spaghetti orders. In the early days, you recruited your chefs by offering each a smartwatch as a sign-on bonus. Unbeknownst to them, you had preloaded the watches with an automatic stopwatch that times each activity they perform, and beams the data to your ~~control center~~ back office.*[18] *Thus, you've amassed histograms for the durations of each of the three cooking activities.*

These histograms are reflected in Figure 2-4, which depicts the temporal network for fulfilling a single order of spaghetti. It's entirely based on your original recipe, except that in the industrial chaos of your restaurant's kitchen, you must account for the occasional extra-ordinary situation: If orders are coming in too fast, you might run out of freshly made meatballs, and have to pull last night's overstock from the freezer, so they'll take longer to bake. Or maybe a previous chef put too much spaghetti in the pot, so instead of boiling a new batch, you just take what's there already. These situations are all covered by your histograms, which essentially smoothes the uncontrollable duration time windows from the STNU version of your plan, previously depicted in Figure 2-2, and adds tails to them.

[17] Precognition at its finest.

[18] You were always gunning for a career in either the tech sector or espionage. Couldn't decide which.

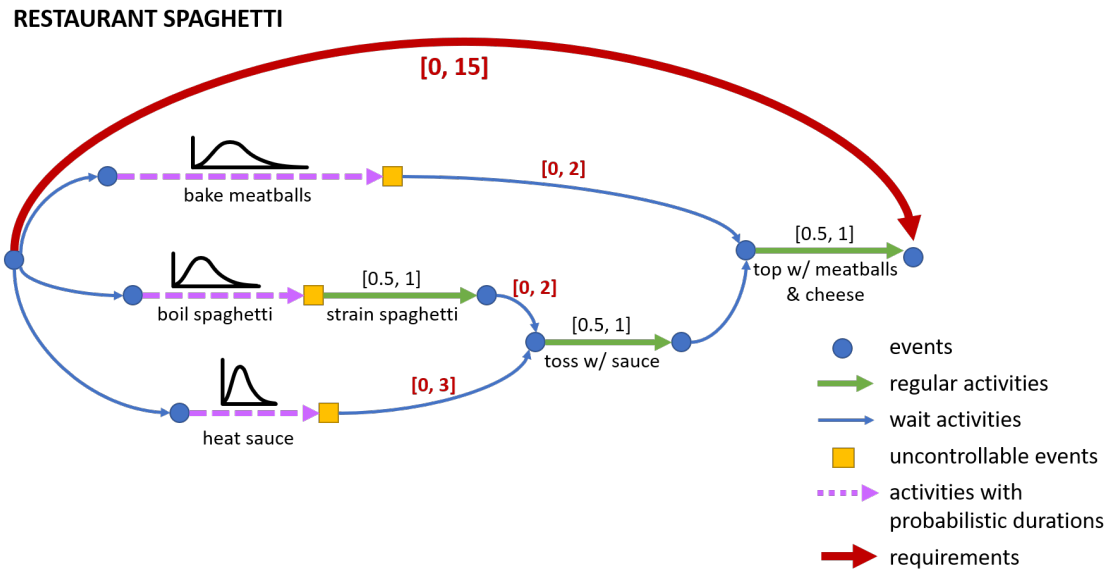

Figure 2-4: This depicts the same temporal network as in Figure 2-2, except you've replaced the time windows for uncontrollable activities' durations with probability distributions.

This example shows that the only real change I am making to the STNU model is adding probabilistic weighting to the uncontrollable activity outcomes. The semantics of executing those activities, controllable or not, remain changed. And therefore, I can reuse all the previous concepts relating to plan execution, including history, outcome, policy, and result function. The only concept modification I need is to assign likelihoods to outcomes.

This modification will turn out to affect the problem statement, such that the original criteria of simply finding feasible policies won't be as meaningful. Instead, a notion of quantifying risk and bounding it will come into play. Stated in the abstract, dealing with risk might seem like a complication that makes the problem harder to solve. But my thesis claim is that framing the problem as bounded risk-taking allows us to safely "cut corners", and thus dispatch plans that might otherwise have been unschedulable. I work towards this problem framing in the rest of this section, and then demonstrate its advantages through experiments in Chapter 7.

Below I present the formal definition of a pSTN, basing it on the STNU as much as possible. Then I establish likelihoods for the outcomes, so that there is a well-defined probability of failing to meet the temporal requirements. From this, it follows that the relevant problem for scheduling pSTNs is not to satisfy the requirements directly, but to

impose some condition on the *probability* of failing to do so.

Definition 2.16 (pSTN)**.** *A* probabilistic STN $\mathcal{N}^p = \langle \mathcal{E}, \mathcal{E}^u, \mathcal{A}, \mathcal{A}^p, \mathcal{R} \rangle$ *follows the same structure as an STNU, except that in place of activities with uncontrollable duration* $\mathcal{A}^u$*, it has activities with* probabilistic *duration* $\mathcal{A}^p$*. Each* $a_j^p \in \mathcal{A}^p$ *takes the form* $\langle e_i, e_j^u, f_j \rangle$*, thus replacing the STNU uncontrollable* $[l, u]$ *bounds with a probability density function* f_j *on the duration of* a_j^p*. The domain of* f_j *is nominally* $[0, +\infty)$*, or a subset thereof.*

By convention, I refer to the cumulative density function as F_j*, i.e.,*

$$F_j(t) = \int_0^t f_j(\omega) \, \mathrm{d}\omega \, .$$

Besides weighting the outcomes of probabilistic durations with likelihoods, the execution semantics of pSTNs are identical to those of STNUs. Therefore, there is still a one-to-one correspondence between each a_j^p *and the uncontrollable event* e_j^u *it ends on.*

Execution-wise, pSTN probabilistic activities behave just like STNU uncontrollable activities, except that they introduce probabilistic semantics, which quantify how much more likely certain outcomes are than other ones. Therefore, I need to update the definition of outcome to account for likelihood. Recall from Definition 2.8 that Nature selects a full outcome ω just before execution. As I hinted on page 49, these semantics make it easier to attach likelihoods, which will be necessary to state the pSTN scheduling problem. Namely, I simply assign likelihoods to the full outcomes, so that distributions for individual outcomes' fall out as marginals of the joint. This is completely general in that it allows for unspecified dependencies between activities' durations. If I had defined the outcome semantics as saying that Nature samples them on-the-spot during execution, then to calculate the full joint distribution (i.e., likelihoods of full outcomes), I would either have to assume all durations are independent, or supply a complex set of conditional distributions.

Definition 2.17 (Outcome likelihood)**.** *For a given pSTN* $\mathcal{N}^p$*, there exists a joint probability distribution* P *over the space of full outcomes* Ω*. I.e, there exists a probability density function* $f : \Omega \to \mathbb{R}$*, such that*

$$\int_\Omega f(\omega) \, \mathrm{d}\omega = 1.$$

Thus, for each probabilistic activity a_j^p, its one-dimensional probability density function f_j is a marginal distribution of the joint f:

$$f_j(\omega_j) = \int_{\Omega \setminus \Omega_j} f(\omega_{\setminus j}) \, \mathrm{d}\omega_{\setminus j} \, .$$

In practice, it's unlikely that you would need to fully specify the joint P (or f) alongside a pSTN. If you're basing your distributions off previous experience (or simulation), you would effectively have a joint distribution from all your previous trials. From those trials, it would be straightforward to create marginal distributions, and thus discard the joint information, in which outcomes for one individual duration were matched with outcomes for another.

With the definition of outcome likelihood in hand, it finally makes sense to talk about the probability, or *risk* of a scheduling policy failing to satisfy the requirements. This wasn't possible with the STNU model, whose scheduling problem required an all-or-nothing approach to feasibility: If there was even a single outcome that caused a policy to produce an infeasible execution history, then that entire policy was deemed infeasible.

With pSTNs, we don't have to be so unforgiving. The definition of feasibility still applies to execution histories, which are fully grounded and therefore have no remaining probabilistic uncertainty. But for a pSTN scheduling policy, we are interested in how likely it is that we would encounter outcomes that lead to infeasible histories. The collective probability mass of those outcomes thus gives us a measure of the "risk of infeasibility" we take when we run that policy.

Definition 2.18 (Failure probability of a policy)**.** *Given a pSTN $\mathcal{N}^p$ and a scheduling policy $\mathcal{P}$ for it, the* failure probability *of $\mathcal{P}$ is the combined probability mass of all the full outcomes $\omega \in \Omega$, where executing $\mathcal{P}$ to completion on ω yields a final execution history ξ_N that is* infeasible*. I.e.,*

$$\mathrm{P}_{\text{fail}}(\mathcal{P}) = \int_{\Omega} \mathbb{1}_{\text{infeasible}}(\xi(\mathcal{P}, \omega)) \mathrm{P}(\omega) \, \mathrm{d}\omega \, .$$

This notion of failure probability is ultimately what my pSTN scheduling problem is about. The final question to answer is what my problem tries to do with that probability, and there are basically two possible answers. I could either find a policy that minimizes it, or I could simply choose to bound it. My answer is the latter, and the reasoning is

that I want to leave open the possibility of choosing other optimization objectives. For example, a common scheduling objective is to finish the plan as fast as possible. This means minimizing the total time between the start and end events *of the plan itself*, also known as the *makespan*. If I choose a policy that minimizes scheduling risk, it may be too conservative in that by taking so much time between activities – in order to buffer against wide ranges of uncontrollable durations – it produces a much longer makespan than expected. Therefore, I have the user explicitly state an acceptable bound on the scheduling risk, so as not to preclude any desire to optimize some arbitrary objective function.[19]

I capture this notion of "acceptable risk" by a *chance constraint*. In a nutshell, a chance constraint allows for a small probability of violating a regular requirement constraint (or a set of them). Chance constraints are an established aspect of planning problems which involve probabilistic uncertainty. For instance, for the problem of path planning with obstacles, in which the vehicle dynamics have probabilistic noise, Ono [46] augments an ordinary qualitative state plan with chance constraints that allow for small, bounded probabilities of hitting certain obstacles. (In addition to hard physical constraints, those obstacles could also represent virtual ones, e.g., a plane trying to avoid no-fly zones or weather boundaries.)

Thus, for pSTNs, I perform a similar augmentation by attaching chance constraints to the temporal requirements. To preface that definition, I want to note that the concept of a policy's failure probability in Definition 2.18 can be specialized on a subset $\mathcal{R}' \subseteq \mathcal{R}$ of the requirements. I.e., I can arbitrarily restrict the scope of (in)feasibility to only certain temporal requirements that I care about in the moment, and ignore the rest. This allows me to have multiple chance constraints over different requirements (or subsets thereof), each tailored with a different levels of acceptable risk. For instance, if it is especially important to me that some activities in the middle of the plan are tightly coordinated, I might ask for a low risk bound on enforcing that coordination, while being more lenient with the plan's overall deadline.

Definition 2.19 (Temporal chance constraint)**.** *Given a pSTN $\mathcal{N}^p = \langle \mathcal{E}, \mathcal{E}^u, \mathcal{A}, \mathcal{A}^p, \mathcal{R} \rangle$, a chance constraint c takes the form $\langle \mathcal{R}', \Delta \rangle$, where $\mathcal{R}' \subseteq \mathcal{R}$ and Δ is a real number between*

[19] In this thesis, I do not include optimization objectives in my problems or algorithms; I only address satisfiability. In my conclusion chapter, I summarize existing and future efforts on integrating optimization into my work, and point out how easy or difficult it might be to deal with certain issues.

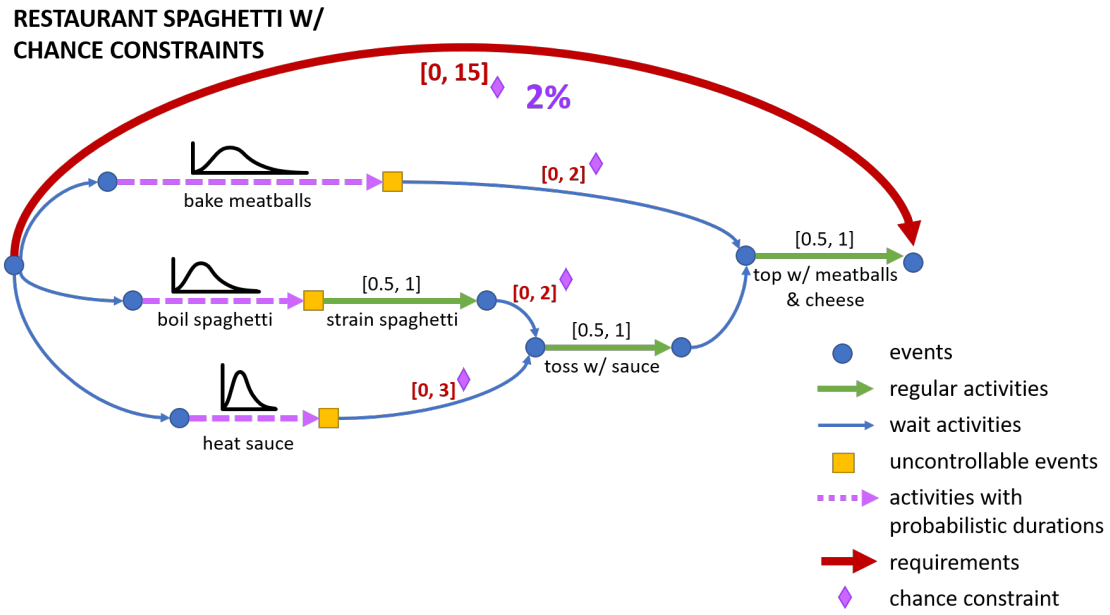

Figure 2-5: This augments the pSTN in Figure 2-4 with a 2% chance constraint on the network's requirement constraints.

0 and 1. It expresses a probabilistic *requirement that a policy $\mathcal{P}$ may not have more than a failure probability of Δ with respect to a subset $\mathcal{R}'$ of the temporal requirements. I.e., with probability at least $1 - \Delta$, the final execution history $\xi(\mathcal{P}, \omega)$ of a policy $\mathcal{P}$ must satisfy all the temporal requirements in $\mathcal{R}'$.*

In this thesis, I limit my scope to handling a single global chance constraint, i.e., a chance constraint on the entire set of temporal requirements $\mathcal{R}$. I do this because addressing multiple chance constraints involves a step prior to my main algorithm that performs an entirely different kind of reasoning. Again, I summarize the issues in my conclusion chapter, and leave it to future work. I still define below my pSTN scheduling problem to allow for multiple chance constraints, but in subsequent chapters, I will assume only a single global one is given.

As the final visual example in this chapter, Figure 2-5 illustrates the application of a global chance constraint to the Restaurant Spaghetti scenario from Example 2.15. The purple diamonds are "attached" to all the temporal requirements, and request that collectively, none of those constraints be violated with more than 2% probability.

At last, I can state my thesis's problem as a *chance-constrained pSTN*. Put simply, we

seek a policy that no longer has to directly satisfy the temporal requirements of a pSTN, but rather has to meet the risk bounds declared in a set of chance constraints over said pSTN.

Problem 2.20 (cc-pSTN scheduling problem)**.** *Given a pSTN* $\mathcal{N}^p = \langle \mathcal{E}, \mathcal{E}^u, \mathcal{A}, \mathcal{A}^p, \mathcal{R} \rangle$*, and a set of chance constraints* $\mathcal{C}$ *over the requirements* $\mathcal{R}$*, find a scheduling policy* $\mathcal{P}$ *that satisfies the chance constraints.*

My final remark is that the same distinction between static and dynamic policies for STNUs applies to pSTN scheduling. This distinction is central to how I organize the subsequent chapters. Namely, while my end goal is to provide chance-constrained *dynamic* policies, my expositional strategy is to aim first for a solution that provides static policies, and then adapt it with the necessary insights for providing dynamic policies.

Chapter 3

Problem Reformulation into Risk Allocation

In the last chapter, I extended the STNU scheduling problem into a chance-constrained version for pSTNs. By introducing probabilistic durations, it became necessary to have a measure of a scheduling policy's risk, i.e., its probability of failure. This risk is difficult to evaluate directly, and therefore makes searching the policy space for a solution quite intractable.

In this chapter, I address that issue by reformulating a cc-pSTN problem into that of *risk allocation*, while leveraging STNU theory in the process. Risk allocation had been previously introduced in the context of probabilistic path-planning [8] [46] to work around the complexity of composing nonlinear probability distributions. The central idea is to distribute the allowable risk throughout the plan intelligently while mapping the original problem down into a non-probabilistic/deterministic formulation. This makes it much more computationally tractable to find solutions, at the cost of some minor incompleteness.

In our chance-constrained scheduling context, risk allocation maps a given pSTN into an STNU. Thus, all the power and efficiency of STNU controllability and dispatching algorithms, including dynamic execution, become available to us. While most prior work in pSTN scheduling has approximated the form of risk allocation, they are not as general or rigorous as the framework that I present. Only [20] and [69] use the same form, and they limit themselves to producing static policies. Fortunately, generating dynamic policies can

happen downstream of risk allocation, so the same framework can be used. Throughout this section, I cast the examples in terms of dynamic execution rather than static when applicable.

This chapter's main contribution, then, over [20] and [69] is a much more rigorous treatment of risk allocation's relationship to the original problem. I mentioned above that risk allocation for path planning introduces some incompleteness. This is an unavoidable feature of the same mechanism that guarantees risk allocation's *soundness*. After explaining in Section 3.1 the sources of the original problem's intractability, I introduce risk allocation in Section 3.2, and give the intuition for its soundness and incompleteness. I also argue that the degree of incompleteness can often be acceptable, but can only be measured empirically, as in Chapter 7.

Finally, Section 3.3 collects these ideas into formal definitions of risk allocation and the reformulated problem that results from casting the original problem as risk allocation. It ends with a systematic accounting of all possible sources of incompleteness, something not previously seen in prior work on pSTN scheduling.

3.1 Intractability of the pSTN policy space

As I established in Definition 2.9, a scheduling policy is a function that maps partial execution histories to scheduling decisions. Therefore, the policy space for a pSTN is a space of functions, which could potentially be quite difficult to navigate.

In general, it's not clear what is the best way to parameterize a policy's behavior. The options are less ambiguous for static policies, because they essentially follow a predetermined execution history $\xi^{\setminus u}$ on the controllable events $\mathcal{E}$. Therefore, you could effectively represent those policies by their backing histories $\xi^{\setminus u}$. That's $N - K$ parameters[1] to consider, one for each controllable event. The ratio between N and K varies between types of plans, and if your plan is chock full of uncontrollable activities, then K might occupy much of N, so you'd have a smaller parameterization. On the other hand, if $|\mathcal{E}^u|$ is small, then you wouldn't be so lucky.

[1]Recall from Chapter 2 that N is the total number of events, and K is the number of uncontrollable ones.

For dynamic policies, parameterization is much more elusive. We'd have to capture the whole range of dynamic decisions these policies could make, i.e., given any partial history, any unexecuted controllable event is a candidate for subsequent dispatch. And because events don't always have to arrive in the same order, the space of partial histories (plus their accompanying unexecuted events) that we'd have to act on is combinatorially huge with respect to N. We could try intuitive strategies that slice the policy space, like, "If event B arrives before $t = 10$, then these policies try to dispatch F anywhere between $t = 12$ and $t = 15$, whereas those policies just `wait`." However, the subsequent divisions you'd need to cover the whole space are just too many. So this is the first challenge of trying to navigate the policy space: that we don't really know how to characterize it in terms of parameters.

Another challenge is distinguishing between policies that are solutions to our problem, and those that are not. Even if we knew how to characterize the policy space, it's not easy to evaluate whether any one spot in it (i.e., any given policy) satisfies our cc-pSTN problem. In the case of STNUs, it's relatively easy to identify the boundaries between feasible and infeasible policies. The reason relates to how STNU policies can be derived out of the structure of the STNU itself.[2] In that process of analyzing the network structure to generate a policy, we only need to consider the lower and upper bounds of the uncontrollable activities' time windows.

For pSTNs, though such reasoning is insufficient. First of all, the effective "time windows" on uncontrollable durations could very well stretch from 0 to $+\infty$, depending on the distributions, so analyzing their bounds is meaningless. Second, we can analyze STNUs by reasoning about durations' lower and upper bounds, because there is no likelihood weighting on the outcomes in between. However, if we did that for pSTNs, we would be throwing away all the distribution information, and therefore be unable to evaluate the

[2]To briefly summarize, the key data structure behind this derivation is the *distance graph* of a temporal network. An STNU is dynamically controllable *iff* its distance graph has no semi-reducible negative cycles. And if so, we can compile the distance graph into a form such that at any point during the STNU's execution, we can update the graph with the latest result, and read out a range of possible next decisions by a feasible policy. I.e., choosing any decision outside the range of those provided by the graph results in an infeasible policy. Therefore, we essentially construct a policy on-the-fly from the distance graph, and it is guaranteed to be feasible due to the properties of the graph.

For strong controllability, we first compile the STNU into an STN, i.e., we compile out the uncontrollable activities and events. Then, we operate on the STN's distance graph, and a similar but simpler theory applies for constructing a static policy.

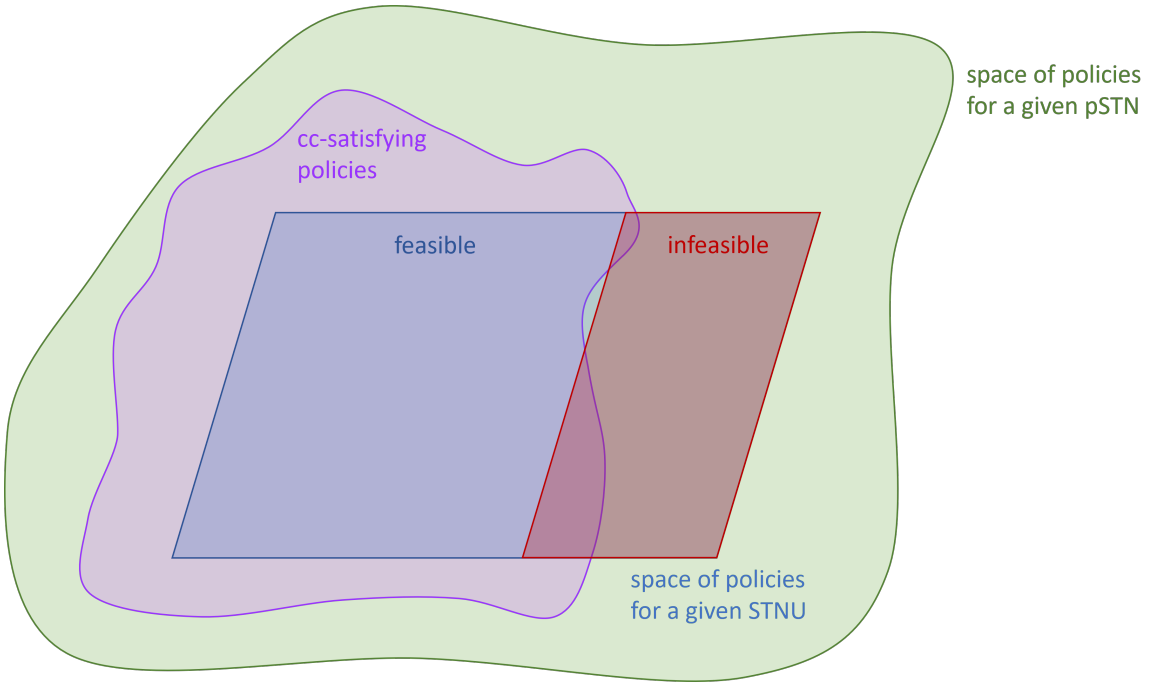

Figure 3-1: The chance constraint of a cc-pSTN problem carves out a well-defined but hard-to-find region (shaded purple) in the policy space. Additionally, it is also difficult to evaluate/certify whether any given policy falls within that region.

probabilistic condition in the chance constraint.

Therefore, the main challenge in deciding whether any given pSTN policy solves a cc-pSTN lies in evaluating the policy's failure probability according to Definition 2.18. I.e., we'd have to integrate the joint probability distribution over the space of outcomes, and for each full outcome, simulate whether the policy succeeds or fails (in meeting the temporal requirements). Only if it succeeds would we include that outcome's probability density in our integration. It should be apparent that such integration over arbitrary nonlinear distributions, with domains potentially stretching to $+\infty$, coupled with the requisite simulation, is a task better suited for massively parallel computing hardware than your ordinary workstation.

Figure 3-1 summarizes these difficulties via a visual approximation of the policy space. Overall in green, we have the entire space of policies for some given pSTN. The wavy boundaries connote that we don't understand how to parameterize or navigate within this space. (Note that this is a space of *functions*, and not the space of *outcomes* for the pSTN. I will discuss the latter in the next section.) Within this policy space lies a region containing all policies which satisfy the chance constraint. While this region is well-defined, its

boundaries are also poorly understood, because it's hard to evaluate whether any given policy actually falls within it (i.e, satisfies the chance constraint).

In contrast, I represent the space of policies for some given *STNU* as a (slanted) rectangular region. The straight boundaries connote that we understand how to build policies from the STNU's structure, so that in effect, we can achieve some sort of parameterization of the policy space. And so, given an STNU policy, we can efficiently analyze it to determine whether it's feasible, without having to simulate it. I show this with a straight-line delineation between the blue and red regions.

Note that I depict STNU policies as a subset of the pSTN policy space. This comes with two caveats: The first is that this implicitly assumes the STNU shares the same *structure* as the pSTN, i.e., how the activities and requirements connect to each other. While the duration models differ, execution histories on the STNU would otherwise be indistinguishable from those on the pSTN. Thus, the act of applying any STNU policy to those histories could be *interpreted* as running a pSTN policy, hence the subset depiction.

This leads to the second caveat, though, which is that STNU activity durations usually have smaller domains than their pSTN counterparts. That means there are technically pSTN outcomes, and hence execution histories, that aren't handled by STNU policies. Fortunately, it is relatively straightforward to define "automatic extensions" to the latter to turn them into proper pSTN policies. I will address this in the next section.

My last point is that I had suggestively oriented the STNU policy space such that its feasible portion falls within the chance constraint region in the overall pSTN policy space. There's no guarantee that this must be the case. However, if we were to find such an STNU, it would be tempting to use any one of its feasible policies as a surrogate for dispatching the pSTN. This hints at the main insight behind how I map the cc-pSTN problem into a more tractable one, which I discuss in the next two sections.

3.2 Leveraging STNU theory

To summarize what we know about STNUs and what we want to accomplish for pSTNs: We know from the literature how to determine if an STNU is controllable. And if it is, we

also know how to obtain a valid scheduling policy. These statements hold for both strong and dynamic controllability, which correspond to static and dynamic policies, respectively.

Our goal is to answer the same questions for pSTNs, but in a chance-constrained context, as established in Problem 2.20. Since STNUs and pSTNs share so much in plan structure and policy structure, it would be reasonable to utilize our knowledge of STNUs to address cc-pSTNs. However, all the reasoning for STNUs is based on the interval bounds that characterize the activities and requirements. (This is a key reason why STNU algorithms are efficient: they explicitly consider only the "edge" cases for each duration, and not all the possibilities in between.) Unfortunately, this means those strategies are not directly applicable to pSTNs' probabilistic durations.

Instead, my strategy to avoid the intractabilities described previously is to map a pSTN's probabilistic durations into STNU interval-bounded form. If we apply this to a pSTN, then the network effectively transforms into an STNU, and suddenly all the power of STNU algorithms is available to us: We verify whether the STNU is controllable, and if so, extract a policy of the desired flavor. As I argued in the previous section, this STNU policy would be applicable to the original pSTN, modulo minor adjustments. I will discuss those adjustments towards the end of this section.

So this strategy is promising in that if we can find such a mapping, then we could get a suitable policy. Except, it doesn't address obeying the chance constraint, which is the actual criteria for our cc-pSTN problem. Therefore, in order to pursue this strategy, the mapping of durations must be responsible for ensuring that whatever policy is produced will respect the chance constraint.

Let's consider what such a mapping would look like on a single probabilistic duration, and what implications it has regarding the chance constraint. Figure 3-2 depicts an $[l, u]$ interval over the probability density distribution f_j of outcomes for a duration. If we had mapped the duration into such an interval and applied STNU semantics, we would be pretending that the duration's possible outcomes can only fall inside that interval.

According to Definition 2.16, though, the original pSTN semantics dictate a reality where the domain of possible outcomes is $[0, +\infty)$. So we would be ignoring any outcomes in the ranges $[0, l)$ and $(u, +\infty)$. Furthermore, the *likelihood* that the actual outcome would

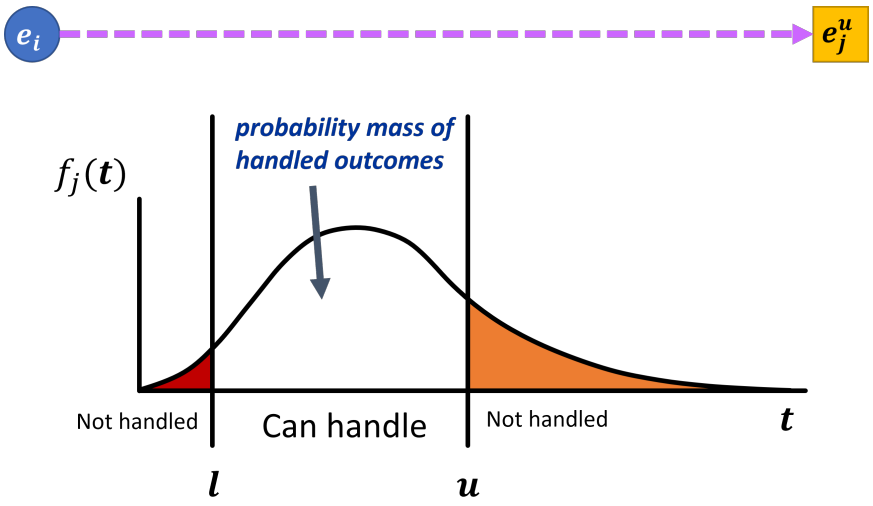

Figure 3-2: By imposing an STNU-style $[l, u]$ interval bound on a probabilistic duration, we ignore any outcomes falling outside those bounds. The *risk*, then, of assuming those bounds is the probability mass of the ignored outcomes. This risk applies to any policy that is based on the $[l, u]$ interval. In other words, if a feasible policy is constructed based on that interval, then the probability mass lying within the interval specifies the likelihood that the policy is guaranteed to handle that duration's outcome.

land in those ranges is given by the probability mass in the tails of the distribution. I call this the *risk* of applying those $[l, u]$ bounds to that duration.

Definition 3.1 (Risk on a duration's outcomes)**.** *Given an activity with probabilistic duration* $a_j^p = \langle e_i, e_j^u, f_j \rangle$*, let the cumulative density function of the distribution be* F_j*. The risk of assuming that the duration's outcome will lie within an interval* $[l, u]$ *is:*

$$R_j = \Pr\left[(\omega_j < l) \cup (u < \omega_j)\right] = F_j(l) + (1 - F_j(u))\,.$$

Therefore, suppose that we construct an STNU by applying some $[l, u]$ bounds to each probabilistic duration. (They don't have to be the same for each duration.) Then with respect to any particular probabilistic duration a_j^p, there is a $1 - R_j$ chance that its outcome ω_j will fall within its STNU bounds. In other words, the STNU is a valid model for $1 - R_j$ of the *pSTN outcomes* for that duration. This condition thus becomes attached to any result derived from the STNU. So if the STNU were controllable, then any of its feasible policies would be applicable to $1 - R_j$ of a_j^p's outcomes.

This is a useful insight about the probabilistic applicability of an *STNU policy* to a

pSTN. Still, though, it doesn't address the exact criteria of our cc-pSTN problem. Namely, we want the policy to hold for a percentage of the *full* outcomes, not just the outcomes for an individual duration. Fortunately, it is straightforward to generalize this insight from being about a single duration to all probabilistics durations in a pSTN. Consider the following example on our restaurant spaghetti scenario.

Example 3.2 (Reframing restaurant spaghetti as an STNU)**.** *Consider the pSTN described in Example 2.15 and depicted in Figure 2-4; call it $\mathcal{N}^p$. As manager and owner of your restaurant, you're very proud of having modeled your situation as a pSTN, but you've been racking your brain ever since trying to find and verify a 2% chance-constrained scheduling policy for it. You pine for the days when you only had to cook spaghetti for your family, which you understood as an STNU (call it $\mathcal{N}^u$; see Example 2.2 and Figure 2-2), and thus could generate policies at ease.*

So in an act of desperation, you cross your eyes and superimpose $\mathcal{N}^u$ on top of $\mathcal{N}^p$. What you get is Figure 3-3. Everything lines up, except the three cooking activities with probabilistic duration have $[l, u]$ bounds slicing through the tails of their distributions.

Let the baking, boiling, and heating activites be indexed by $j = 1$ through 3, respectively. Then it is clear that the STNU would be a valid model for the pSTN's outcomes (and thus execution) when each duration's outcome falls within its interval bounds. This happens with probability:

$$\Pr\Big[(10 \leq \omega_1 \leq 14) \bigcap (8 \leq \omega_2 \leq 11) \bigcap (2 \leq \omega_3 \leq 4)\Big].$$

Equivalently, the probability that some outcome will not *fall outside its interval bounds, and thus invalidate the STNU as a proxy for the pSTN, is*

$$\begin{aligned} R^u = \Pr\Big[&(\omega_1 < 10) \cup (14 < \omega_1) \bigcup \\ &(\omega_2 < 8) \cup (11 < \omega_2) \bigcup \\ &(\omega_3 < 2) \cup (4 < \omega_3)\Big]. \end{aligned} \tag{3.1}$$

R^u represents the risk of using $\mathcal{N}^u$ as a proxy for executing $\mathcal{N}^p$. In other words, any

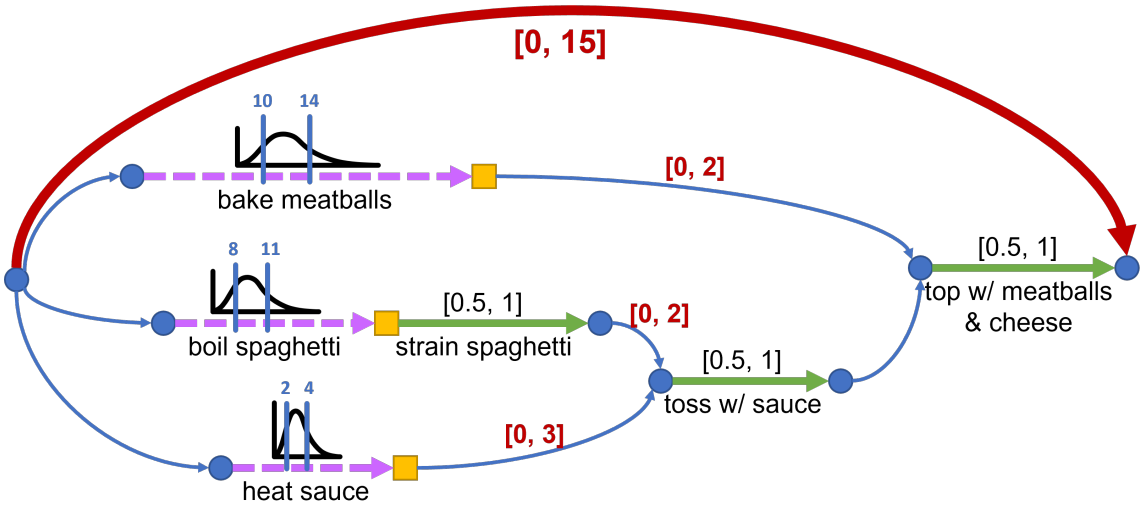

Figure 3-3: Imposing STNU-style $[l, u]$ bounds on all the probabilistic durations of a pSTN, yields an STNU which carries the *risk* of ignoring any outcomes falling outside those bounds.

feasible policy for $\mathcal{N}^u$ is also a feasible policy for $\mathcal{N}^p$ with probability at least $1 - R^u$.

This example brings up a key observation: Before, I reasoned that the risk R_j of assuming $[l, u]$ bounds on a single duration a_j^p carried over to any policy that depends on those bounds. But we can apply same reasoning to the STNU $\mathcal{N}^u$ that results from assuming such bounds on all probabilistic durations simultaneously. Equation 3.1 expresses of doing so, and hence that risk gets attached to using any feasible policy of $\mathcal{N}^u$ for dispatching the original pSTN $\mathcal{N}^p$.

Now recall that the chance constraint (Definition 2.19) requires our policy to have a failure probability no larger than Δ. If we can ensure that R^u is no larger than Δ, then any feasible policy for $\mathcal{N}^u$ will automatically satisfy the chance constraint. This is very promising! It means we have mostly decoupled the reasoning about meeting the risk bound Δ and the reasoning about satisfying the temporal requirements $\mathcal{R}$. The former is handled by calculating R^u based on the $[l, u]$ bounds, and the latter is supplied by STNU controllability theory. These two aspects of reasoning intersect only in that they must be based on the same $[l, u]$ bounds, and together, they guarantee the existence of a policy that satisfies the chance constraint.

STNU controllability and policy generation is a solved problem, so I will not dwell

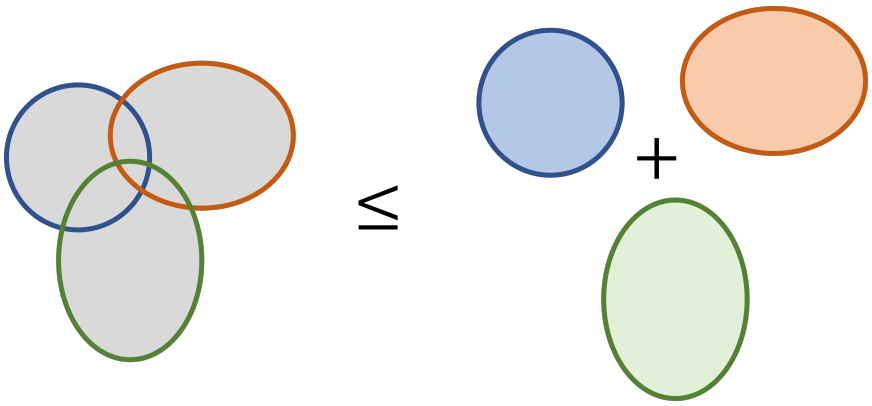

Figure 3-4: Illustration of the union bound. The probability mass of the union of distinct events is upper-bounded by the sum of the probability masses of the individual events.

on it here. Rather, the key remaining question is how to calculate R^u efficiently. The probability expression in Equation 3.1 is difficult to evaluate because the probabilistic event in question spans multiple random variables. In fact, the total event is a multivariate union, and since the semantics of outcome likelihood (Definition 2.17) allow for arbitrary full joint distributions, there is little hope of evaluating its probability directly. In particular, I do not assume that the marginal distributions for individual durations are independent.

My key insight to overcome this hurdle is to note that the total event is a *disjunction of single-variable events*. Specifically, the probabilistic event in Equation 3.1 is a disjunction of pairs of events for ω_1, ω_2, and ω_3. The probability of each of those pairs is just the individual duration risk defined in Definition 3.1. Namely, it's the probability mass of that duration's marginal distribution that lies outside the $[l, u]$ bounds. These marginal distributions are supplied for each probabilistic duration as part of the pSTN (Definition 2.16), and their cumulative density functions are directly accessible.

Recall that the goal is to upper-bound R^u by Δ. Therefore, at this point, I invoke the union bound (also known as Boole's inequality) to split Equation 3.1 into probabilities of individual single-variate events. For reference, Figure 3-4 depicts the union bound on three events A, B, and C, which may overlap with each other on certain (full) outcomes. The probability of the union of all three events is upper-bounded by the sum of the probabilities of the individual events.

Applying the union bound to R^u yields:

$$\begin{aligned} R^u = \Pr\Big[&(\omega_1 < 10) \cup (14 < \omega_1) \bigcup \\ &(\omega_2 < 8) \cup (11 < \omega_2) \bigcup \\ &(\omega_3 < 2) \cup (4 < \omega_3)\Big] \\ \leq \quad &\Pr\left[(\omega_1 < 10) \cup (14 < \omega_1)\right] \\ &+ \Pr\left[(\omega_2 < 8) \cup (11 < \omega_2)\right] \\ &+ \Pr\left[(\omega_3 < 2) \cup (4 < \omega_3)\right] \\ = \; &R_1 + R_2 + R_3. \end{aligned}$$

Therefore, as long as we enforce

$$R_1 + R_2 + R_3 \leq \Delta, \tag{3.2}$$

then we will also guarantee $R^u \leq \Delta$.

Equation 3.2 is much more tractable to evaluate, because unlike R^u, we are able to evaluate the individual R_i's directly. This condition is called a *risk allocation*, because we are effectively taking a risk budget of Δ and "distributing" it additively across the individual sources of uncertainty. Risk allocation had been previously introduced by Blackmore [8] and Ono [48] in the context of path planning with obstacles. In our pSTN context, distributing risk R_i to a probabilistic duration a_i^p means cutting off its distribution's tails at l_i and u_i, such that combined mass in those tails is exactly R_i. More risk means more probability mass in the tails, so the $[l_i, u_i]$ interval would shrink in width.

Figure 3-5 illustrates the concept of risk allocation on our example, and in the next section, I will formalize it. Note that it's possible Equation 3.2 might not allocate all of Δ to the durations. Thus there may be "leftover risk" not allocated to any duration, and this is okay. It just means we're being more conservative than necessary, i.e., the uncontrollable intervals are wider than they need to be. If there were leftover risk and we allocated it to some durations, it would only shrink those durations' intervals. If the STNU implied by

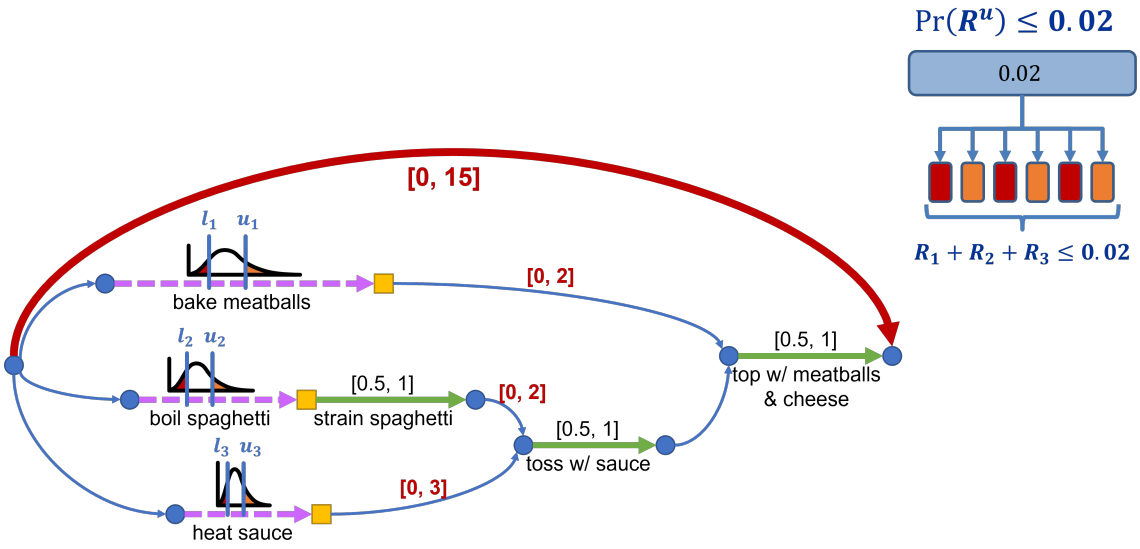

Figure 3-5: Applying risk allocation to the spaghetti pSTN. Rather than enforcing the risk bound on R^u, which is hard to evaluate, we enforce it on the sum of the individual risks. This can be interpreted as distributing a "budget" of Δ risk over the individual durations.

the original intervals is already controllable, then allocating any remaining risk would still preserve controllability.

While the conservatism of having leftover risk is harmless, it does raise the issue of what kinds of conservatism are present in the process of problem reformulation and risk allocation. Recall that I proposed this approach to gain tractability into finding policies for the original cc-pSTN. Along the way, I argued that mapping the pSTN into an STNU and applying the risk allocation constraint guarantees that any policy for the STNU will respect the chance constraint. Therefore, this process is *sound*. However, the converse is not true, i.e., there may exist feasible scheduling policies for the original cc-pSTN that this approach cannot find.

In the next section, I formally discuss *incompleteness* in terms of the relevant solution spaces. Here, though, I present the intuition first, using the example below.

Example 3.3. *Figure 3-6 shows a very simple pSTN with two probabilistic durations in sequence. An overall constraint requires the two activities to finish between a and b time units after the start. The two durations d_1 and d_2 are modeled as Gaussians with means μ_1 and μ_2, and standard deviations σ_1 and σ_2. Their joint outcome space is depicted on the right. The purple ovals indicate the contours of the joint Gaussian distribution. (The*

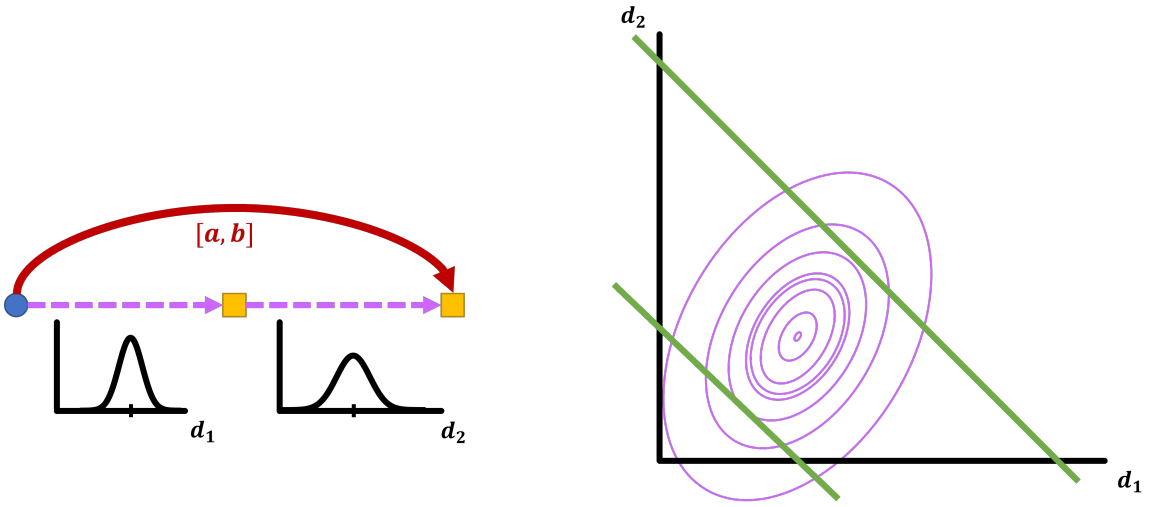

Figure 3-6: A very simple pSTN with two probabilistic durations in sequence, plus an overall deadline. There is only one trivial scheduling policy, which is to dispatch the start, and let the rest of the plan execute. The space of duration outcomes is two-dimensional, and the policy succeeds on all outcomes in the trapezoidal region between the two diagonal lines.

contours get closer where the gradient of the PDF is larger.) Note that I've indicated there is some dependence between the two durations, given the tilt of the distribution's axes. However, when projected/marginalized onto the individual durations, we recover the one-dimensional Gaussian PDFs that come with the pSTN.

Technically, these Gaussians should be truncated below $t = 0$ and renormalized. However, if the standard deviations are significantly smaller than their respective means (e.g., $\sigma \leq \frac{\mu}{5}$), and the chance constraint bound is, say, on the order of 1%, then the probability mass below $t = 0$ is negligible.

By inspection, we can see that there is only one trivial policy: dispatch the controllable start event (say at time 0), and let the two activities run in sequence with whatever durations Nature picks. The question then is with what probability will the $[a, b]$ requirement be satisfied? In this simple example, we can easily visualize the exact answer: The condition that $a \leq d_1 + d_2 \leq b$ is represented by the trapezoidal region in between the two green lines, which intersect both axes at a and b. The sum $d_1 + d_2$ has a distribution that is the convolution of d_1's and d_2's. And if you accept that the probability masses of the Gaussians below 0 (i.e., outside of the first quadrant) are negligible, then $d_1 + d_2$ also follows a Gaussian

(you'd be projecting the multivariate along a diagonal axis perpendicular to the green lines). You could then integrate that Gaussian from a to b (equivalently, integrating over the area of the corridor between the green lines) to get the exact success probability.

Now let's see what conservatism/incompleteness is present in the STNU reformulation and risk allocation approach. On the left of Figure 3-7, we see a rectilinear region carved out by the $[l_1, u_1]$ and $[l_2, u_2]$ bounds. These are the outcomes captured by the STNU we've transformed the pSTN into. This region needs to be contained within the green boundaries in order to guarantee satisfaction of the $[a, b]$ requirement, and thus controllability of the STNU.

Therefore, the largest likelihood of success we can guarantee is the largest probability mass we can cover with a rectilinear region that fits in the corridor. I.e., suppose the exact success probability was p^*, but we could only find an STNU that covers $p' < p^*$. Then whereas in theory there exists a policy that meets a chance constraint with risk bound $\Delta^* = 1 - p^*$, the best we can do is meet a risk bound of $\Delta' = 1 - p'$, which is larger than Δ^* by $p^* - p'$

Whether this is incompleteness is costly depends on how "wide" the joint distribution is relative to the width of the corridor. For the purpose of illustration, this example stretches out and orients the joint distribution so that it's clearly difficult to capture most of the mass in the corridor with an STNU projection. However, when there are many probabilistic durations, we get a joint distribution in a high-dimensional space, and the immense "volume" of these spaces means most of the mass tends to be tightly clustered. So for large plans, we'd expect to be better able to capture the mass with a rectilinear prism that represents the STNU's outcome space.

We also have to consider the conservatism introduced by the union bound approximation in risk allocation. Recall that rather than evaluating the success probability, i.e., the mass inside the rectangle, we are evaluating the risk R^u of being outside. Equation 3.2 approximated and upper-bounded R^u by adding the tail masses of the individual marginals. This is depicted on the right side of Figure 3-7. The tail masses for duration d_1 are in the two vertical strips of red outside l_1 and u_1, and for d_2 are in the horizontal strips outside l_2 and u_2.

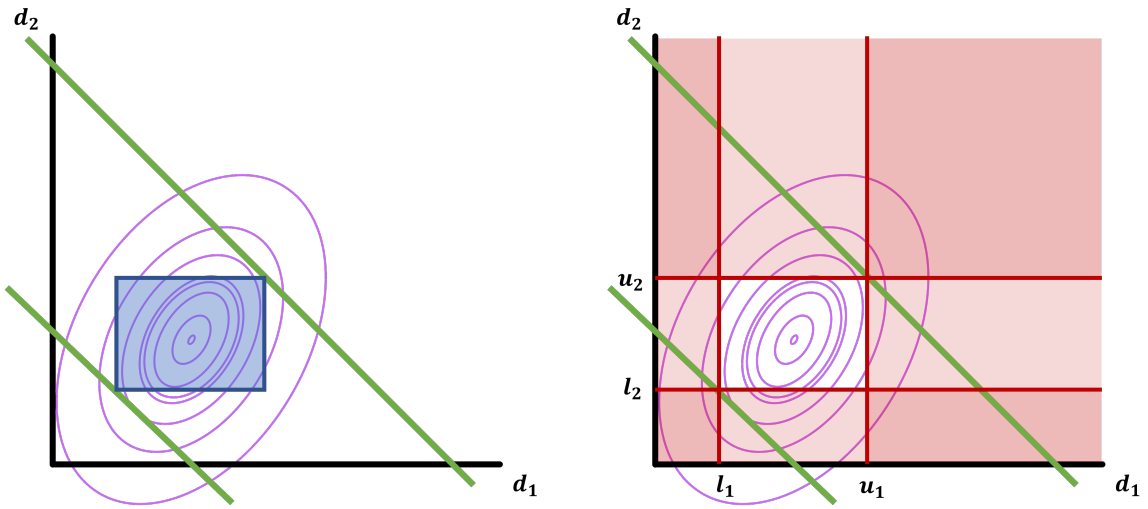

Figure 3-7: Transforming the pSTN into an STNU prevents us from collecting the full set of joint outcomes that would lead to feasibility, since we don't perform convolution. Then, the union bound that risk allocation relies on creates further conservatism through overlap / double-counting of joint outcomes that the STNU doesn't capture.

These four strips overlap with each other in the doubly-shaded regions. Therefore, risk allocation is double-counting the probability mass in those regions, and thus further increasing the lowest risk bound we can guarantee. For larger plans, this would lead to even higher-multiple counting of certain regions.

Fortunately, there is good reason to believe this conservatism isn't costing us too much probability mass, either. As Figure 3-7 suggests, the probability masses in the overlapped regions are already quite small. In this example, the two durations are not independent, but they're not super-dependent on each other, either. Thus, if the tails we cut off from the individual marginals are already small, then their combined effects in the overlapping regions result in very little mass there.

When Ono [46] presented risk allocation for path planning, he identified the worst-case conservatism as $\frac{N-1}{N}\Delta$, where N was the number of distributions to which he was allocating risk. (In the pSTN case, it would be the number of probabilistic durations.) However, he also showed that if all durations were independent, then the probability mass we give up is $O(\Delta^2)$. Namely, as we enforce tighter risk bounds, the conservatism decreases *by an additional factor of* Δ. This means as long as we are interested in chance constraints with small Δ's, the union bound provides a very good approximation of R^u.

Even though I presented the pSTN model without assuming independence, we can ask how close actual durations are to being independent, and thus how often we achieve this good approximation. This in turn depends on the scenario at hand. In a public transit context, if a bus takes longer to drive between stops due to traffic, then there's some reason to believe that the same traffic would similarly affect the bus immediately following. However, subway cars don't have to deal with road traffic, so their travel durations are more likely to depend on driver behavior, which is less tightly coupled.

Generally, activities that are closer in time or happening in parallel have more opportunities to have dependent durations. Larger separations in space and/or time afford fewer opportunities for underlying processes to link activities. Therefore, for large plans that span significant time, it's reasonable to expect mainly local dependencies, and hence we shouldn't do much worse than the $O(\Delta^2)$ conservatism when there's perfect independence.

To summarize this section, I introduced a two-step reformulation of the original cc-pSTN problem to make searching for policies tractable. First, I cast the problem as searching for a proxy STNU that is controllable, so that we can use any of its policies to dispatch the pSTN. Those policies are guaranteed to work with probability at least the joint distribution mass covered by the STNU's outcome space. However, evaluating the joint mass, is still difficult. So the second step is to recognize the complement of that mass represents the total risk of assuming the STNU model, and that the union bound can upper bound the total risk by the sum of the individual duration risks, which are accessible. This effectively reformulates the chance constraint, which I will formally state in the next section.

I also explained how both of these steps introduce their own form of conservatism, so the reformulation is sound but incomplete. I argue, though, that the incompleteness can be tolerable under many circumstances. Again, in the next section, I will more formally document the sources of incompleteness.

The last item I promised to address in this section is to explain how to adapt an STNU policy to be applicable for pSTN execution. Right after the definition of pSTNs (Definition 2.16), I noted that with the exception of outcome generation, the execution semantics for pSTNs were identical to STNUs. This means that the STNU definitions of scheduling policy (Definition 2.9) and results of policy decisions (Definition 2.10)

apply equally for pSTNs. Namely, pSTN policies still map partial histories into decisions, and Nature decides whether those decisions get preempted by the termination of active probabilistic durations.

However, towards the end of Section 3.1, I noted that while probabilistic durations in pSTNs are defined over the domain $[0, +\infty)$, their uncontrollable duration counterparts in STNUs have finite $[l, u]$ domains. Therefore, for a pSTN and an STNU of the same structure (i.e., activities and requirements), there are many more possible execution histories for the pSTN than the STNU. Any policies for the STNU technically don't have defined behavior for execution histories containing outcomes beyond their respective durations' $[l, u]$ domains. So, this poses a slight hurdle for our strategy of using STNU policies as proxies for dispatching pSTNs.

Fortunately, it is easy to patch this discrepancy. I already argued that once we transform a pSTN into an STNU, any feasible policy for an STNU will behave *exactly the same* on the pSTN *for those outcomes*. Furthermore, if those outcomes cover at least $1 - \Delta$ of the probability mass (due to our risk allocation strategy), then the STNU policy already nominally satisfies the chance constraint. That means we don't really care how the policy behaves on the remaining outcomes. We just have to define some behavior so that the policy will execute the plan to completion, rather than producing "undefined" scheduling decisions.

In other words, for these exceptional cases, we can effectively ignore the temporal requirements $\mathcal{R}$, and just focus on executing the remaining activities of the plan. The main condition, then, is to make sure the policy still generates decisions leading to *valid* execution histories, rather than fully *feasible* ones. I.e., the policy has to respect the duration semantics of the activities at all times, whether or not unhandled outcomes have occurred.

There are a couple possible strategies for defining such policy behavior. For example, you could try to complete the rest of the plan as quickly as possible, by executing each of the remaining controllable activities with their shortest allowed duration. (Nature still gets to decide the durations for any probabilistic durations.) But you could also choose any duration within those activities' flexible $[l, u]$ intervals, and the execution history would still be valid.

The only problematic case is if you are trying to dispatch a wait activity (recall, these are the blue arrows in the example diagrams, *not* the `wait` decision that could be output by a policy), the upper bound is $+\infty$. For example, consider the merging of the "bake meatballs" and "toss w/ sauce" activities in Example 2.15 and Figure 2-4. Choosing $+\infty$ for either of the waits' durations would force the other one to be $+\infty$ as well, and thus cause execution to hang forever.

Therefore, to propose a valid, deterministic strategy for converting STNU policies into valid pSTN ones, Definition 3.4 goes with the shortest possible duration for each remaining activity. This covers all scenarios except when two or more threads of execution are merging via wait constraints.

To use the above example of merging "baking" and "tossing", suppose the meatballs finished baking at $t = 10$, but the other cooking activities took too lone, so we couldn't finish the tossing activity until $t = 13$. Then we'd have no choice but to wait at least 3 minutes after "baking". Then according to our strategy, we'd execute the wait activity from "tossing" to "topping" with its shortest duration of 0 minutes. Thus, we'd begin "tossing" at $t = 13$, simultaneously ending the wait activity from "baking". Note that this violates the $[0, 2]$ requirement on that wait activity, but we've already thrown that out since either the "boil spaghetti" or "heat sauce" activities took too long.

Definition 3.4 (Extension of STNU policy into pSTN context)**.** *Let $\mathcal{N}^p$ be a pSTN, and let $\mathcal{N}^u$ be an STNU with the same structure, but with probabilistic durations mapped into finite-domain uncontrollable durations. For every probabilistic duration a_j^p in $\mathcal{N}^p$, and let it's STNU counterpart a_j^u have domain $[l_j, u_j]$. Let $\mathcal{P}^u$ be a scheduling policy for $\mathcal{N}^u$, and let ξ_k be the current execution history that $\mathcal{P}^u$ has produced on the pSTN $\mathcal{N}^p$ so far. The current time t_k is the execution time of the last event e_{i_k} in the history ξ_k.*

If according to ξ_k, the following two conditions are true, then $\mathcal{P}^u$ is still a valid policy for $\mathcal{N}^p$, so we continue to use it to get the next decision σ_{k+1}.

1. *Any probabilistic duration a_j^p that has completed had an outcome ω_j that fell in the range $[l_j, u_j]$.*

2. *For any probabilistic duration a_j^p that is still active (i.e., has been dispatched but*

not yet terminated by Nature), the current time is compatible with it's mapped STNU domain. I.e., suppose a_j^p began at $t = t_j$. Then $t_k - t_j \in [l_j, u_j]$.

Otherwise, we switch to the following strategy:

1. *First, check if any free-hanging controllable events that begin threads haven't been executed. (E.g., in the spaghetti example, suppose we didn't have the initial start event. Then the three cooking activities would have free-hanging events that start their respective threads.) Decide to schedule any one of them (or equivalently, all of them) right now (at time t_k).*

2. *For each controllable-duration activity a_i currently in execution, determine the time t_i it started according to the history ξ_k, and let it's flexible time window be $[l_i, u_i]$. We are guaranteed that the current time has not exceeded a_i's latest possible time, i.e., $t_k \leq t_i + u_i$.*

3. *According to the activity model, the remaining valid window of execution for a_i would be $[\max(t_k, t_i + l_i), t_i + u_i]$. Let $t_i^* = \max(t_k, t_i + l_i)$, which is the earliest time we could decide to terminate a_i.*

4. *Pick the activity with the smallest t_i^*. If that activity's end event does not merge two or more threads, then there is no coordination needed to make sure those threads end together. Therefore, decide to terminate that activity at the time of its t_i^*.*

5. *Otherwise, if the activity does merge with other threads, then it must be a wait activites, with a $[0, +\infty)$ interval, and so must the other activities merging with it. As the spaghetti example above illustrated, merging can only happen when all wait activities are in execution. So we need to check if all the other wait activities are active. If so, we can immediately merge and terminate all those waits right now at time t_k (which t_i^* must be equal to). Otherwise, we are not ready to merge, and we move on to the activity with the next smallest t_i^*.*

6. *Finally, if all threads have been started, and the only activities in execution have probabilistic duration, then the only possible decision is to* `wait` *for one of their uncontrollable end events to arrive.*

While this definition may seem long, it really just expresses the simple idea of dispatching a plan to completion as soon as possible without regard for the temporal requirements. All the technicalities are needed to ensure that the policy respects the model of controllable activity durations, and thus produces valid execution histories. These principles are rooted in the original dispatching strategies for STNs and STNUs. For more information about dispatching, see work by Muscettola, Morris, and Tsamardinos [43] [58] [40] as well as by Hunsberger [33].

The ultimate takeaway, though, is that Definition 3.4 closes the technical gap between a plain STNU policy and adapting it for use with a pSTN. The definitional extension only applies when some duration's outcome falls outside of the STNU model. There is always a possibility the resulting execution history may still satisfy the temporal requirements, but we make no effort to do so, nor any attempt to quantify the likelihood of that happening. To simplify the exposition in subsequent sections and chapters, I will assume this extension applies automatically to any STNU policy.

3.3 The cc-pSTN risk allocation problem

The previous section described the intuition of mapping the original pSTN down into an STNU and viewing it as a form of risk allocation. This section will now formally state those concepts and use them to transform the original cc-pSTN problem (Problem 2.20) into a reformulated problem that is much more tractable to solve.

I begin by defining what a risk allocation is for a cc-pSTN. Applying a given risk allocation to a pSTN yields an implied STNU that is a proxy for the pSTN, and we wish the STNU to be controllable. The reformulated problem thus reduces to finding an appropriate risk allocation, which must satisfy two conditions: The first is that the risk allocation respects the original chance constraint's risk bound Δ. I call this condition a reformulated chance constraint. The second condition is for the implied STNU to be controllable.

At the beginning of Chapter 4, I will review how others have derived constraints that express the condition of STNU controllability. The derivations are rather involved, so I will not repeat them, but I will summarize their form and the computational complexity

of deriving them. Combined with the reformulated chance constraint, they will form a mathematical programming encoding of the reformulated problem. Thus, this will show that the reformulated problem is tractable, i.e., one could employ off-the-shelf solvers to find risk allocations for reformulated cc-pSTNs. However, I will also argue that the encoding is not as efficient as could be, and that will motivate my algorithmic approach in Chapter 4.

For now, in this chapter, my concern is with defining the reformulated problem, and stating its relationship to the original problem. Namely, I have argued previously that this reformulation approach is sound but incomplete. I will end this section by documenting the possible sources of incompleteness. However, this is for theoretical completeness, and I have given the intuitions for why these sources would not be overly conservative.

Now, let's begin with the central concept of risk allocation. Recall from Figure 3-5 that the idea is to distribute a Δ risk bound over the individual probabilistic durations, such that their individual risks R_i sum to at most Δ. In turn, for a duration to take on R_i risk means to cut off tails from that durations' distribution such that their probability masses sum to exactly R_i. Therefore, a risk allocation can be represented by a set of $[l, u]$ intervals imposed on the probabilistic durations, and the CDFs for those distributions allow us to express or verify that the tail masses we're ignoring do not exceed the desired risk bound.

Definition 3.5 (Risk allocation)**.** *Given a pSTN $\mathcal{N}^p = \langle \mathcal{E}, \mathcal{E}^u, \mathcal{A}, \mathcal{A}^p, \mathcal{R} \rangle$ and a chance constraint $c = \langle \mathcal{R}, \Delta \rangle$, a* risk allocation *$\mathcal{L}$ is a mapping from $\mathcal{A}^p$ to $[l, u]$ bounds. Each probabilistic duration $a_j^p \in \mathcal{A}^p$ receives the bounds $\mathcal{L}(a_j^p) = [l_j, u_j]$, and according to Definition 3.1, this leads to an individual risk R_j on a_j^p's duration. The overall risk allocation $\mathcal{L}$, then, must satisfy*

$$\sum_{j=1}^{K} R_j \leq \Delta, \tag{3.3}$$

where $K = |\mathcal{A}^p|$.

The first key concept in risk allocation is expressed in Equation 3.3, which is a linear inequality in R_j. However, the actual variables we control are the l_j and u_j bounds. As Definition 3.1 says, R_j is derived from these bounds based on the CDF F_j of a_j^p's distribution, and generally F_j has nonlinear characteristics. (The only case in which it would be fully linear is if the distribution was uniform.) Therefore, the actual condition of

satisfying the risk budget is nonlinear in terms of l_j and u_j, and I call this the *reformulated* chance constraint.

Definition 3.6 (Reformulated chance constraint)**.** *Given a pSTN $\mathcal{N}^p$ and a chance constraint $c = \langle \mathcal{R}, \Delta \rangle$, the* reformulated chance constraint $\bar{c}$ *is a condition on finding a risk allocation $\mathcal{L}$ for the cc-pSTN. Namely, if $\mathcal{L}$ maps each probabilistic duration a_j^p to an interval $[l_j, u_j]$, then we require:*

$$\sum_{j=1}^{K} [F_j(l_j) + (1 - F_j(u_j))] \leq \Delta, \tag{3.4}$$

So this takes care of the original chance constraint: If a risk allocation $\mathcal{L}$ satisfies this condition, then any policy $\mathcal{P}$ we find based on $\mathcal{L}$'s assumptions will satisfy the original Δ risk bound. The next question is how we find that policy, and the answer lies in the intuition expressed at the beginning of Section 3.2: By assuming that the probabilistic duration outcomes actually fall within the $[l_j, u_j]$ bounds specified by $\mathcal{L}$, we transform the pSTN model semantics into those of an STNU. Thus, a risk allocation conceptually maps a pSTN into an implied STNU.

Definition 3.7 (Implied STNU)**.** *Given a pSTN $\mathcal{N}^p$, a chance constraint c, and a risk allocation $\mathcal{L}$, the* implied STNU $\mathcal{N}^u$ *is a temporal network that is identical to $\mathcal{N}^p$, except that each probabilistic duration a_j^p has been turned into simply an uncontrollable one a_j^u whose range of possible durations is $\mathcal{L}(a_j^p) = [l_j, u_j]$.*

All we need now is for the implied STNU to be controllable (either strongly or dynamically), and that will guarantee the existence of a feasible scheduling policy $\mathcal{P}$ (which we then extend into a pSTN policy according to Definition 3.4). The fact that $\mathcal{P}$ is feasible for the STNU, combined with the reformulated chance constraint condition, makes it a valid solution for the original cc-pSTN problem.

There is just one wrinkle, which is although we have efficient (i.e., low-order polynomial-time) algorithms for checking STNU controllability [41] [38] [39] and then executing scheduling policies for them [33], these algorithms all operate on *fully grounded* STNUs, where the bounds on the uncontrollable durations are constants. What I've stated above is we require the *condition* that the STNU is controllable, and we are trying to assign $[l_j, u_j]$ bounds such that this is true.

Fortunately, since STNU controllability is well-defined, it is thus also well-defined to state such a condition on finding such bounds. As I've said, I will delay until Chapter 4 elaborating on how to encode that condition in terms of the l_j and u_j variables we have to assign. However, it is good news that others have worked out constraint encodings of this precise condition, both for strong controllability [60] [20] and dynamic controllability [66] [13]. This lends further evidence that this condition is well-defined.

At last, I can state the reformulation of our original cc-pSTN problem (Problem 2.20) in terms of the risk allocation we are looking for.

Problem 3.8 (Reformulated cc-pSTN risk allocation problem)**.** *Given a pSTN $\mathcal{N}^p$ and a chance constraint c, find a risk allocation $\mathcal{L}$ for the cc-pSTN such that the reformulated chance constraint $\bar{c}$ is satisfied, and the implied STNU is controllable.*

While the original problem is about finding a scheduling policy for the pSTN, the reformulated problem only asks for a risk allocation. This is because once a suitable risk allocation is found, the implied STNU will have fully grounded (i.e. constant) intervals for its uncontrollable durations. Existing algorithms can thus directly dispatch the STNU, effectively running a policy.

By imposing this risk allocation structure on our solution, the reformulated problem explicitly decouples the temporal and probabilistic aspects of the original problem. Before, if we had tried to evaluate a candidate policy $\mathcal{P}$ for $\mathcal{N}^p$, we would have had to identify the set of outcomes on which $\mathcal{P}$ yields feasible histories, and simultaneously determine the collective probability mass of those outcomes. The issues with that direct approach, as discussed in Section 3.1 and then illustrated in Figure 3-6, are that set has complex boundaries, and it is difficult to integrate an arbitrary joint distribution over it.

Risk allocation avoids these complexities by considering only rectilinear subsets of the outcome space that could correspond to an STNU. It is easy, though still nonlinear, to approximate the probability of outcomes landing outside such a region, and to ensure it meets a risk bound. It is also straightforward, even if not concise, to then separately express that the STNU must be controllable.

Figure 3-8 summarizes these two decoupled conditions given by Problem 3.8. The

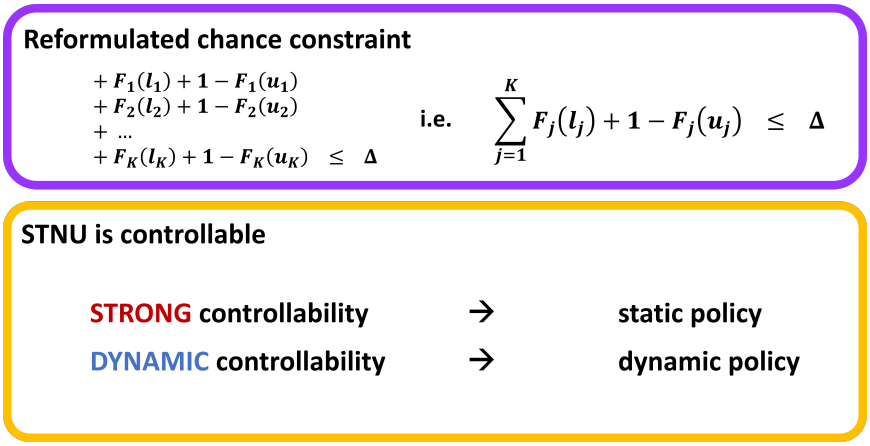

Figure 3-8: The reformulated risk allocation problem expresses two decoupled conditions on the desired risk allocation. First, it must satisfy the reformulated chance constraint. Second, the implied STNU needs to be controllable. Whether it is strongly or dynamically controllable determines whether we thus have a static or dynamic policy for the original cc-pSTN.

reformulated chance constraint handles the risk bound, while the controllability conditions specify desired structure of the implied STNU in order for us to get a policy out of it. We choose which flavor of controllability to specify depending on whether we are solving for a static or a dynamic policy.

Throughout this chapter, I have argued that this approach produces valid policies that respect the original chance constraint, and therefore the reformulated problem is sound with respect to the original. I also gave an example in the last section, proving that limiting ourselves to these rectilinear subsets makes the approach incomplete. In the following subsection, I systematically document the sources of incompleteness.

3.3.1 Incompleteness of risk allocation

To be complete with respect to solving a problem, an algorithm or approach needs to be able to find a solution if a solution actually exists. Therefore, completeness with respect to a cc-pSTN is ultimately about what *portion* of the entire set of chance constraint-satisfying policies our reformulated problem captures. This set was depicted in purple in Figure 3-1, with wavy boundaries connoting the nonlinear probabilistic condition of satisfying the chance constraint.

For comparison, I also depicted what an STNU's policy space might look like, as a subset of the pSTN's. A feasible STNU policy makes decisions based on linear conditions on the event execution times [33], so the boundaries of that region would be "faceted". Thus, if we find a risk allocation solution, we have no reason to believe the implied STNU's feasible policy space lines up with the original full set of chance-constrained pSTN policies.

There is more to the story, however. Unlike Figure 3-1, we are not looking at just one STNU; we are considering an entire space of possible STNUs, implied by the space of risk allocations. Of course in the end, we will settle on a single grounded STNU (or none at all), but we should consider the space where it comes from. We should also consider the space of duration outcomes, both for the pSTN and the STNU, since the notion of risk derives meaning from the joint distribution over that space, and what portion of the pSTN's outcome space the STNU covers. (This was shown on a concrete example in Figures 3-6 and 3-7.)

To summarize, then, I will consider three spaces: the space of pSTN and STNU policies, the space of risk allocations, and the space of pSTN and STNU outcomes. Tracing the logic of the reformulated approach, I will first make the connection between the risk allocation space and the outcome space. Then I will link the outcome space to the policy space. Along the way I will identify where the approach of mapping a pSTN into an STNU introduces incompleteness/conservatism.

To begin with, the reformulated problem asks us to find a risk allocation. Since I defined a risk allocation as a mapping from probabilistic durations to $[l_j, u_j]$ bounds, one can view the l_j and u_j as the parameters of interest. Therefore, for a pSTN with K probabilistic durations, the risk allocation space has $2K$ dimensions. This is depicted in the upper right of Figure 3-9.

Now, since every risk allocation maps to an implied STNU, we can consider that STNU's outcome space. Following the logic of Example 3.3 and Figure 3-7, that outcome space will be a rectilinear polytope subset of the pSTN's outcome space. This is shown in the lower left of Figure 3-9. (High-dimensional polytopes are difficult to visualize, so I've represented it as a convex polygon with parallel sides.) The risk R^u of using the STNU as a proxy for the pSTN is the probability mass falling outside that polytope. This was

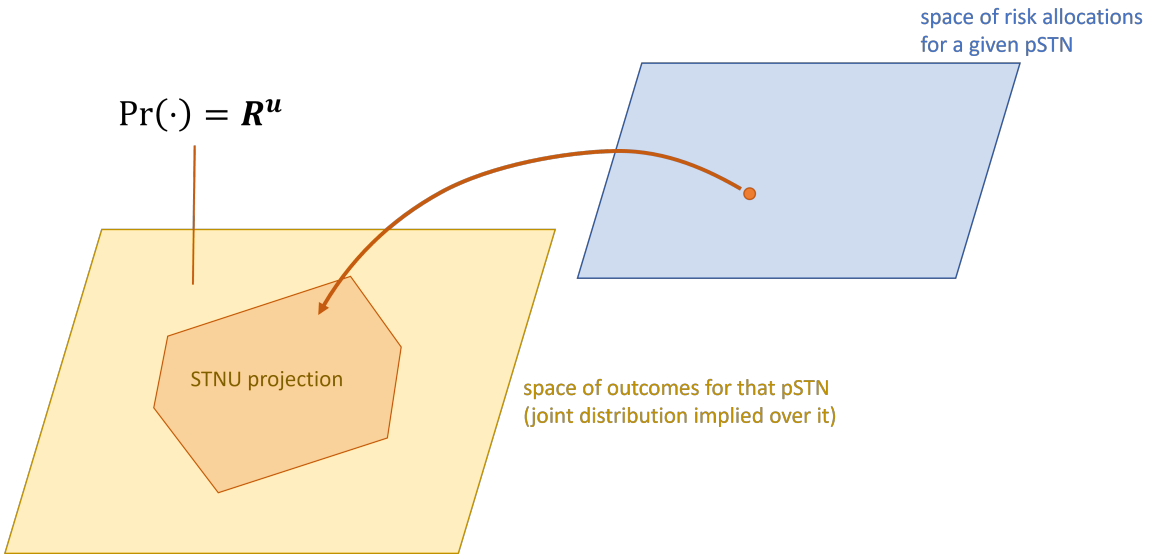

Figure 3-9: Each risk allocation maps to an STNU whose outcome space is a (rectilinear) polytope subset of the entire pSTN outcome space. The probability mass lying *outside* the STNU's outcomes is the STNU's total risk R^u.

illustrated in Example 3.2 with Equation 3.1.

Since the goal is to have $R^u \leq \Delta$, this carves out a desired region in the risk allocation space. However, it's hard to evaluate the (probabilistic) disjunctive event that R^u represents, so our actual reformulated chance constraint applies the union bound, which tightens the constraint to $\sum R_j \leq \Delta$. We had shown in Example 3.3 and Figure 3-7 that this tightening makes the actual risk R^u smaller than necessary. Therefore, the reformulated chance constraint is a subset of the desired region. Figure 3-10 updates Figure 3-9 to show this relationship. We thus see the conservatism introduced by the union bound: that we can't access every risk allocation whose true total risk satisfies the risk bound.

Inside the risk allocation space, there should also be the second condition of the implied STNU being controllable, which would carve out another feasible region. (Generally, risk allocations with very wide $[l_j, u_j]$ bounds would yield uncontrollable STNUs, and thus be likely to fall outside the region.) So, the set of solutions for the reformulated problem would be the intersection of the reformulated chance constraint region and the controllability region. I have not depicted this second region, because the constraints that form it have been proven sound and complete with respect to the definition of STNU controllability [42] [30] [33]. Therefore, it would not impact completeness with respect to

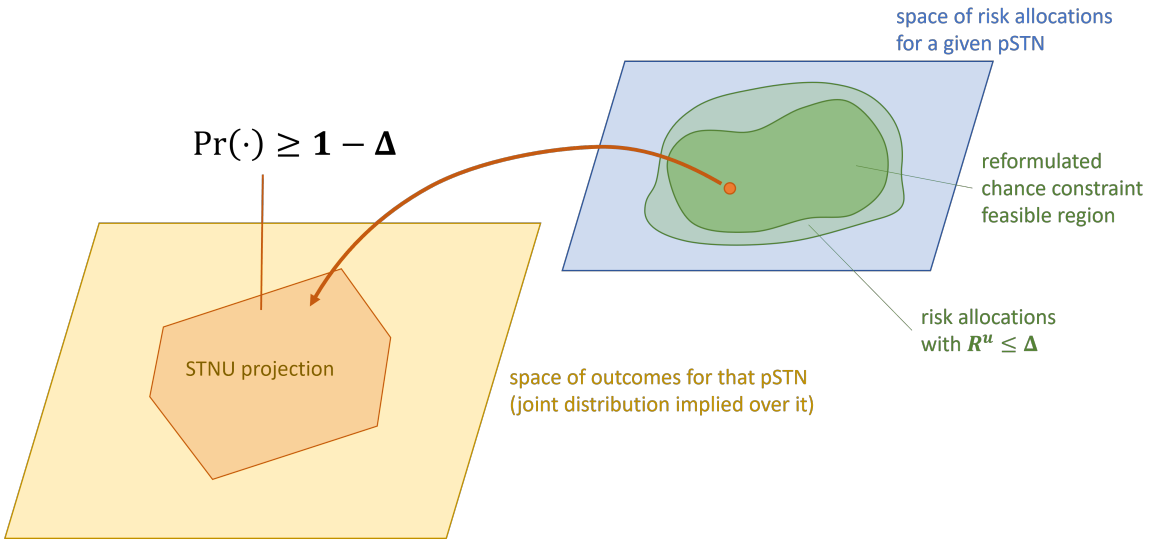

Figure 3-10: There exists a set of risk allocations (shaded light green) whose total risk R^u is no larger than Δ. The reformulated chance constraint (shaded darker green) then applies the union bound, which tightens the region. Thus, by enforcing the reformulated chance constraint, we are unable to discover potential risk allocations solutions.

the cc-pSTN problem.

Having identified a risk allocation and the outcome space of its implied STNU, we can ask what the STNU's policy space looks like. As I discussed in Section 3.1 and also referenced at the beginning of this subsection, the STNU's policy space is a subset of the pSTN's. This is because a) policies for the STNU are receiving execution histories on the same plan structure, but b) the STNU contains a subset of the pSTN's outcomes. Thus, the STNU's policies don't have to handle as many potential execution histories.[3] Figure 3-1 had shown exactly that.

However, if the STNU's total risk doesn't exceed Δ, i.e., the chosen risk allocation falls within the light green region, then *all feasible policies* for the STNU must satisfy the original chance constraint. Figure 3-11 depicts this by showing the blue region of feasible STNU policies falling entirely within the purple chance constraint region. This diagram, then, illustrates two additional sources of incompleteness.

[3]One might question this argument since we are automatically extending all STNU policies into valid pSTN policies according to Definition 3.4. However, the extension is applied *after the fact* to cover a technicality, and it provides only a single deterministic strategy for responding to any outcome outside the STNU's model. Therefore, there are still many pSTN policies that could respond differently (e.g., executing later than at the earliest possible time), which our STNU policy extension does not express.

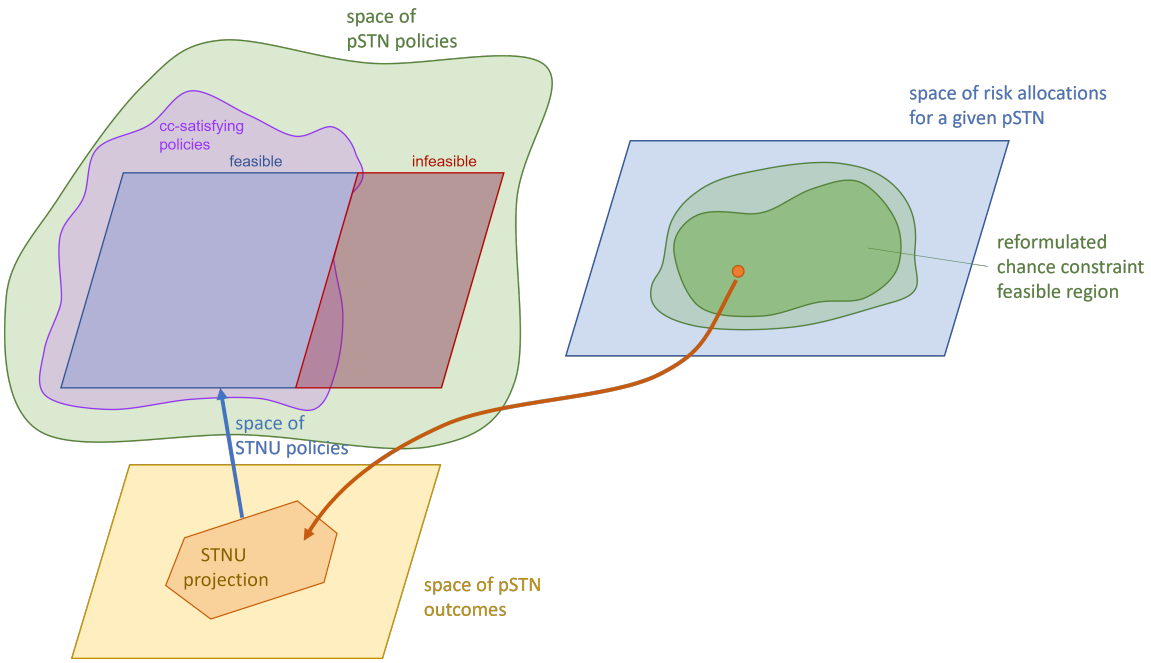

Figure 3-11: Given an STNU implied by a risk allocation, the STNU's *policy space* is also a subset of the pSTN's policy space. If the risk allocation satisfies the $R^u \leq \Delta$ condition, then the feasible subset of the STNU's policy space must fall within the set of chance constraint-satisfying policies.

First, we see there are chance constraint-satisfying policies not covered by the STNU feasible set. The reason why had been hinted at in Example 3.3. Even though that scenario admitted only one possible policy (i.e., dispatch the start event and just keep waiting), it demonstrated that if we were to extract the policy from the implied STNU, that policy would only have been guaranteed to handle the outcomes in the blue rectangular region in Figure 3-7. Namely, the STNU policy would not explicitly address any outcomes outside that region.

In that specific example, it didn't matter since there was only one possible policy. But let's modify the scenario by appending in sequence a controllable activity, shown in Figure 3-12. Now the policy space is enriched by the decision of when to terminate that activity.

Suppose we perform risk allocation and get our STNU with $[l_1, u_1]$ and $[l_2, u_2]$ bounds. We then choose a policy that always decides to run the last activity for 7 time units. If we run that policy on the pSTN, but then encounter an outcome where, say, $d_2 < l_2$, then the

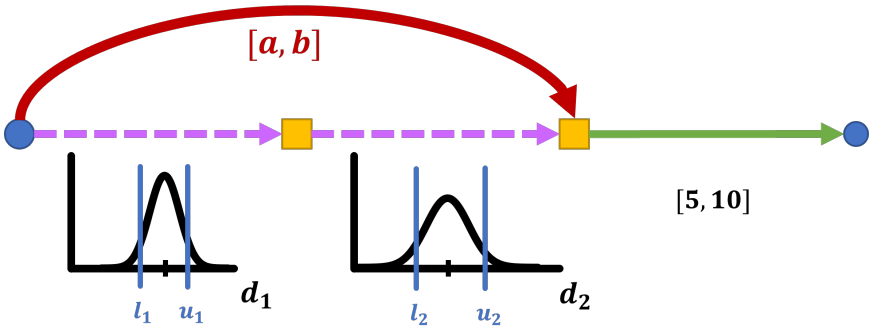

Figure 3-12: This extends the temporal plan in Example 3.3 by a single activity with controllable duration interval $[5, 10]$, so that the policy has to decide for how long to execute that activity.

STNU policy defaults to the extension, which would run the last activity for only 5 time units.

In contrast, a pSTN policy would be able to run the activity for 7 times units no matter the previous outcomes. And if the $[a, b]$ requirement were "wide enough" relative to the combined distribution of $d_1 + d_2$, then that policy would satisfy the original chance constraint. Thus, it would fall inside the purple region but not the blue in Figure 3-11. This proves that if we find a risk allocation that yields a controllable STNU, it is possible for there to exist chance constraint-satisfying policies that are not among the STNU's policies.

The second source of incompleteness shown by this diagram is that there may be *infeasible* policies for the STNU that turn out to be in the chance constraint-satisfying region. This is harder to come up with an example for, so I will just explain the concept. If a policy is infeasible for the STNU, then all it means is there is at least one full outcome on which it doesn't succeed. It might still succeed on the vast majority of the STNU's outcomes. *And*, once the pSTN extension is applied, it might *coincidentally* succeed on some of the remaining pSTN outcomes. The probability mass of those outcomes could outweigh the STNU outcomes that the policy doesn't work on, and thus, the (extended) policy would still satisfy the original chance constraint. But that policy would never be considered, at least in the context of the risk allocation that gave us the implied STNU.

Lastly, I relate feasible STNU policies back into the space of *pSTN* outcomes. Suppose we have such a policy as depicted in Figure fig:stnu-policy-outcome-space. We already know that it will succeed on the entire STNU outcome space. But it may also, by chance, succeed

on some remaining portion of the pSTN outcomes. This was illustrated in Example 3.3 and Figure 3-7, where the (only) policy would succeed on the trapezoidal region between the two green boundaries. If the blue region inside the trapezoid already contains enough probability mass to meet the chance constraint, then the policy's actual success probability will be higher than $1 - \Delta$.

So as Figure 3-13 shows, every STNU feasible policy may succeed on a larger portion of the pSTN outcomes than just the STNU's outcomes, and therefore has a potentially smaller than Δ chance of violating temporal requirements. Hence, this is yet another another form of conservatism, i.e., that we don't consider all the pSTN outcomes that an STNU policy succeeds on. This means there could be risk allocations that don't satisfy the reformulated chance constraint condition, and so their implied STNU's outcomes have too little probability mass. And yet, one of the STNU's policies might succeed on enough additional outcomes to meet the chance constraint's $1 - \Delta$ requirement.

In Example 3.3, the joint distribution could be too spread out and/or the chance constraint too tight for there to exist a rectilinear region within the trapezoid, containing enough probability mass. But as long as the trapezoid itself has enough mass, then the STNU policy would still be a solution to the original problem. Yet, our reformulated problem would not be able to find it because the reformulated chance constraint region in the risk allocation space would be empty.

It is worth noting that the argument for this last form of conservatism is also an argument for the soundness of risk allocation. Namely, since the portion of outcomes an STNU policy succeeds on encompasses the STNU's own outcomes, and since risk allocation guarantees those STNU outcomes to have at least $1 - \Delta$ probability mass, then the policy directly satisfies the original chance constraint's semantics.

To summarize, I have identified four forms of conservatism in my risk allocation approach for the reformulated problem:

1. The union bound approximation of an STNU's risk prevents us from accessing risk allocations that would otherwise satisfy the chance constraint's risk bound.

2. For any implied STNU we use as a proxy for the pSTN, our policies can't explicitly

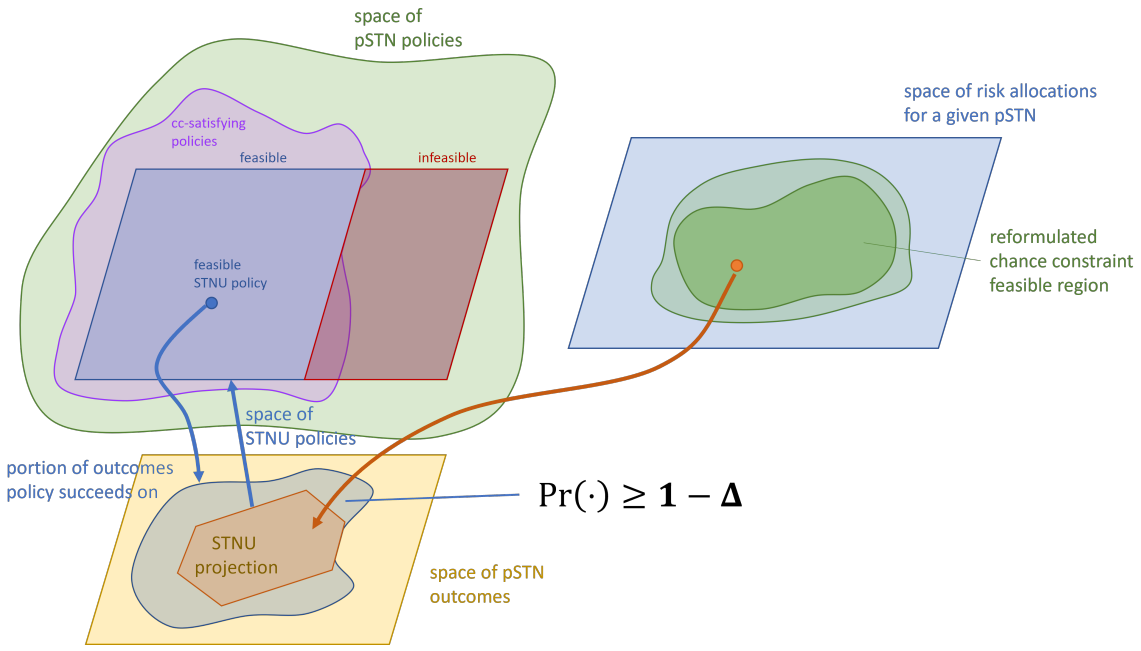

Figure 3-13: Given an STNU implied by risk allocation, any feasible policy we select from the its policy space will succeed (i.e., respect the temporal requirements) on all the STNU's outcomes. But it may also succeed by coincidence on other pSTN outcomes (after applying the pSTN policy extension in Definition 3.4).

react to outcomes not modeled by the STNU, and there may be policies with such abilities that satisfy the original chance constraint.

3. For any implied STNU, there may be infeasible policies that we don't consider but which also satisfy the original chance constraint.

4. For any implied STNU, its feasible policies may succeed on more outcomes that just those modeled by the STNU.

I have also given numerous explanations of risk allocation's soundness. These results are officially stated below.

Lemma 3.9 (Incompleteness)**.** *Problem 3.8 is incomplete with respect to the original cc-pSTN problem 2.20.*

Lemma 3.10 (Soundness)**.** *Problem 3.8 is sound with respect to the original cc-pSTN problem 2.20.*

This has been a theoretical discussion of where conservatism/incompleteness could creep in. In Section 3.2, I had argued that these sources were unlikely to significantly reduce the solution space. Moreover, some incompleteness is expected when trying to make tractable continuous probabilistic conditions that have been composed in a complex constraint network. While one could always construct pathological examples, the loss of solution space ultimately depends on the actual scenario at hand.

Chapter 4

Conflict-Directed Approach to Risk Allocation

Chapter 3 recast our original cc-pSTN problem as a risk allocation instead, such that any solution implied by a valid risk allocation would satisfy the original, too. The key feature of this reformulation was to separate out two classes of constraints, one probabilistic and one temporal, illustrated in Figure 3-8. We fully specified the probabilistic condition, namely, the reformulated chance constraint in Definition 3.6. However, we deferred specifying the temporal controllability conditions until this chapter.

I begin by showing that if we did encode those conditions directly, then it would be expensive to combine them with the reformulated chance constraint and solve for a risk allocation. The key takeaway is the form of those constraints, and the complexity of expressing them. While the encoding for strong controllability is somewhat tractable, the encoding for dynamic controllability is notably more complex. There are also significant hurdles to solving the DC constraints in conjunction with the reformulated chance constraint.

This chapter's contribution, then, is to provide a more efficient, alternate approach to solving the reformulated problem's conditions. To do so, I leverage previous work in chance-constrained *static* scheduling [67] [69], which used a conflict-directed architecture in lieu of fully encoding strong controllability. Here, I adapt that architecture to produce *dynamic* policies instead.

The key aspect that's affected in the conflict-directed approach is the kind of conflicts

we're dealing with. In this approach, risk allocations are generated *without* fully specifying controllability. Rather, they are checked for satisfying those conditions, and if they don't, a *conflict* is produced that informs subsequent rounds of risk allocation. If relatively few rounds are needed to find a controllable STNU and thus a satisfying policy, then the hypothesis is that this approach will be faster than solving the full set of controllability conditions at once.

The existing version of this architecture discovered *strong* controllability conflicts and specified how to resolve them. That meant the final implied STNU would only be guaranteed to be strongly controllable, and so we could only output static policies. In this chapter, I generalize conflict discovery and resolution to include *dynamic* controllability, so we have a choice of whether to output static or dynamic policies.

The key difference with DC conflicts is that instead of being simple linear conditions in the risk allocation space, they are *conjunctive* linear constraints. In turn, that means their resolutions are *disjunctive* linear constraints, which can be interpreted as convex polytopes obstacles in the risk allocation space. I defer until Chapter 5 to show how to reason around those polytope obstacles. Here, my focus is on deriving the form of those constraints and showing how they plug into the high-level architecture.

I review the previous conflict-directed approach, but as with Chapter 3, I present it in a more general context so that the extension to dynamic policies builds naturally on the prior concepts. At the end of this chapter, I also rigorously address soundness and incompleteness, which prior work had not discussed. Namely, I show that the conflict-directed approach preserves the soundness of the problem reformulation and does not add additional conservatism. In other words, this approach is guaranteed to solve the reformulated Problem 3.8 exactly.

4.1 Inefficiency of directly encoding risk allocation

In Section 3.3 and Figure 3-8, I had separated out the reformulated chance constraint and the task of specifying the controllability constraints. Now, I address the latter, which has been already performed by others in both strong [60] [20] and dynamic [13] [66] controllability

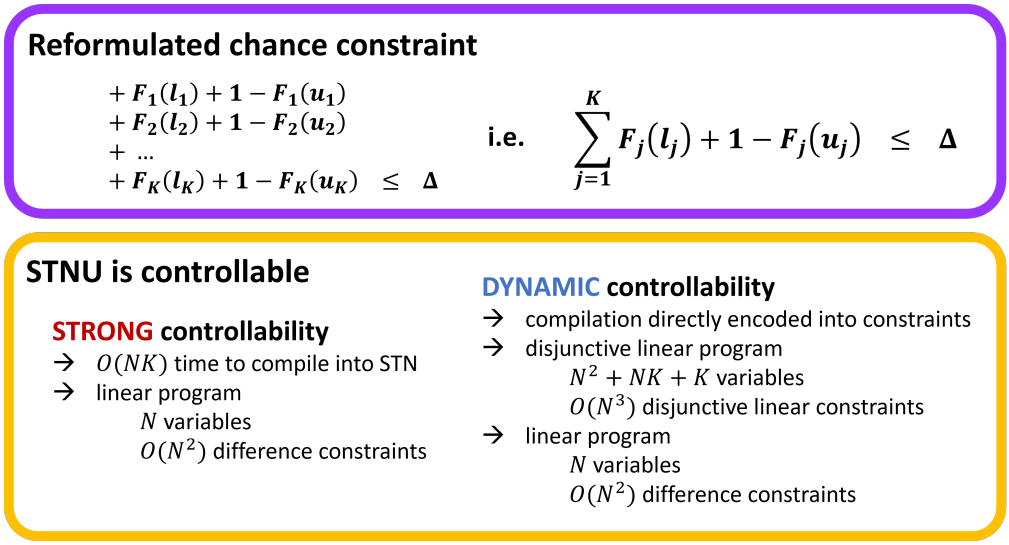

Figure 4-1: Strong controllability can be encoded by statically compiling the STNU into an STN and then encoding the STN as a linear program. Dynamic controllability requires the addition of $O(N^2)$ variables and $O(N^3)$ *disjunctive* linear constraints, thus forming a disjunctive linear program, on top of the STN LP. When combining these conditions with the nonlinear reformulated chance constraint, it's significantly easier to use a black-box constraint solver for the SC conditions than the DC ones.

contexts. My goal here is to summarize the computational difficulties if we directly apply those constraint encodings when solving our reformulated problem.

Figure 4-1 lists the key information needed to show why this approach would be inefficient. It elaborates on Figure 3-8 by stating the complexity of simply writing down the the constraints in the controllability encodings. The following subsections detail the key insights needed to arrive that these complexity results. For now though, I focus on the effect trying to solve constraints of those sizes and form.

In the case of strong controllability, we basically compile the STNU form into an STN. This removes any scheduling dependence on the outcomes of the uncontrollable durations, which is the essence of strong controllability. For every controllable activity and requirement in the original STNU, if either of its endpoints "touches" an uncontrollable event, it gets compiled into a new constraint that only touches controllable events. These new constraints effectively form the STN and express the condition of STN consistency.

Now since each activity or requirement in an STN specifies an $[l, u]$ time window between two events, it can be viewed as a pair of difference constraints over event variables.

Therefore, if the STNU has N events and M activities and requirements altogether, then the strong controllability encoding consists of $2M$ linear constraints over at most N event variables. It's quite reasonable to assume $M = O(N^2)$, because even if there were multiple activities/requirements linking the same pair of events, we only need to consider the tightest conditions. For plans where the number of requirements does not greatly exceed the number of activities, the constraint graph can be considered sparse, that is, $M = O(N)$. Nonetheless, in the worst case, we have a linear program of $O(N^2)$ constraints over $O(N)$ variables.

Combining this LP with the reformulated chance constraint was the approach of Fang [20]. Recall that the $[l, u]$ bounds on each probabilistic duration are the risk allocation variables. Thus, for a pSTN with K such durations, there are $2K$ such variables. Those variables actually get baked into the $2M$ linear constraints of STNU strong controllability during compilation. However, we must have $K \leq M$ and $K \leq N$, so this falls within the $O(N)$ variable count.

The size of the resulting nonlinear program is not unreasonable, and generic nonlinear programming solvers, such as Snopt [25] or Ipopt [63], would be applicable. However, it was demonstrated by Wang [69] that a conflict-directed approach, which avoids solving $O(N^2)$ constraints all at once, produces about an order-of-magnitude (10x) runtime improvement. Therefore, it is natural to ask if a conflict-directed approach to producing *dynamic* policies would yield similar improvements over solving a direct encoding of dynamic controllability plus the chance constraint.

To answer this question, we have to first understand what that direct encoding would be and its effect on the constraint program. Like strong controllability, the encoding of dynamic controllability also includes consistency of a compiled STN. This is accounted for in Figure 4-1 with a final LP. However, that compilation cannot be performed "offline", because DC semantics impose conditions on the derivation of intermediate constraints. Namely, those intermediate constraints can only be derived from other existing constraints, original or derived, that meet certain conditions. That means we cannot *unconditionally* compile each STNU activity or requirement into a final STN form, like we do with strong controllability.

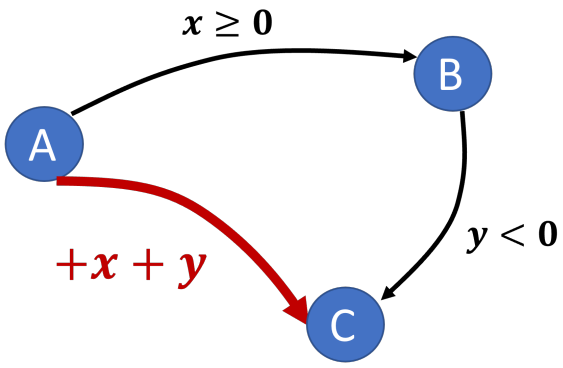

Figure 4-2: Dynamic controllability *conditionally* derives constraint AC if AB's and BC's weights satisfy the "plus-minus" relationship.

Therefore, the compilation semantics must be baked into the DC encoding itself. This was first recognized by Wah [66] and further generalized by Cui [13]. The idea is that since DC semantics specify the conditional derivation of $A \rightarrow C$ from $A \rightarrow B$ and $B \rightarrow C$ in the constraint graph, we can express this logically.

Formally, the derivations operate on what's known as the *distance graph* of the STNU. For each activity or requirement with start and end events A and B, the distance graph contains a forward edge $A \rightarrow B$ and backward edge $B \rightarrow A$, each with a numeric weight. Let x and y be the weights of edges $A \rightarrow B$ and $B \rightarrow C$, respectively. Then we can *conditionally* impose a bound on the weight z of edge $A \rightarrow C$ as follows:

$$(x \geq 0) \wedge (y < 0) \Rightarrow (z \leq x + y). \tag{4.1}$$

This is shown in Figure 4-2.

The form of Equation 4.1 is a logical implication whose clauses are linear inequalities. Recall that implications can be rewritten as disjunctions, i.e., $a \Rightarrow b$ is equivalent to $\neg a \vee b$. Equation 4.1 can be thus be rewritten as

$$(x < 0) \vee (y \geq 0) \vee (z \leq x + y). \tag{4.2}$$

Hence, these additional constraints can be understood as disjunctive linear constraints.

The actual form and semantics of these edge derivation constraints are slightly more

nuanced, and we discuss them in Subsection 4.1.2. But the point is that such constraints can be expressed over all triples of events, which can include both controllable and uncontrollable events. This is what gives rise to the $O(N^3)$ disjunctive linear constraints listed in Figure 4-1.

Furthermore, this means we need to introduce variables to represent the *edge weights* in the distance graph. Some of those will be the original edge weights, corresponding to the $[l, u]$ bounds in the STNU. (And recall that in our implied STNU, the $[l, u]$ bounds on uncontrollable durations are variables, while the rest are constants.) But we need variables to represent derived edge weights as well, so we actually need variables to represent all possible edges.

It might seem that N^2 edges are enough, given that there are N events total, but these N^2 actually only correspond to what are called ordinary edges in the STNU distance graph. In fact, there are *three* types of edges in an STNU distance graph: ordinary edges, lowercase-labeled edges, and uppercase-labeled edges. Again, we elaborate in Subsection 4.1.2, but for now, we just need to know there are always K lowercase edges, and up to NK uppercase edges.

What we've learned then, is that the DC encoding requires, on top of the SC encoding, $O(N^3)$ constraints of the form given by Equation 4.1 or 4.2, over $N^2 + NK + K$ edge weight variables. Thus, the DC encoding both enlarges the number of constraints and variables by a polynomial degree, *and* raises the complexity of the constraints by requiring disjunctive linear constraints. These constraints form a *disjunctive linear program* (DLP), and most solvers are not equipped to handle the combination of that with our nonlinear reformulated chance constraint. At the very least, any inefficiencies in the direct encoding approach to generating static chance-constrained policies would only be magnified when switching to the DC encoding in order to obtain dynamic policies.

One could consider existing approaches to solving the DLP and consider whether it's possible to graft on the nonlinear constraint. For instance, Li [36] offers a branch-and-bound algorithm for solving DLPs. This applies principles of combinatorial search and propagation over the disjunctions, and an LP solver is called at the leaves of the search. In our case, we'd have to swap in an NLP solver, which is significantly more expensive. Also,

we'd have $O(N^3)$ disjuncts, which would correspond to the nominal depth of the search tree. The branch-and-bound strategy coud help reduce this, but ultimately, the number of leaves to consider is exponential in the search tree's actual depth.

Cui [13] offers a mixed-integer linear programming (MILP) reformulation of the disjunctive linear constraints. The reformulation introduces extra constraints and variables, but allows them to take advantage of powerful MILP machinery in MILP solvers such as CPLEX or Gurobi. Unfortunately, generalizing this to mixed integer *nonlinear* programming (MINLP) becomes very expensive.

Finally, Wah [65] [66] introduced a nonlinear programming encoding of the DC semantics. Although it also introduces extra constraints and variables, these are cleverly arranged so that no integer variables or disjunctive constraints are required. Therefore in principle, the same NLP solver could be used to handle them along with our nonlinear chance constraint.

The downside is that this encoding is particularly complex, and also involves nonconvex constraints. Cui [13] clarified certain aspects of this encoding, and showed in experiments that it is both slower and solves fewer problem instances than their MILP approach. In addition to the complexity of encoding, it also suffers from fact that NLP solvers are designed to step locally through a continuous space, and thus may inefficiently explore unpromising areas. Considering how NLP solvers perform on the static policy version of our reformulated problem [69], it is not promising to consider adding $O(N^3)$ *more* constraints, along with the multiplicative factor of extra constraints and variables per original constraint.

In conclusion, directly encoding STNU dynamic controllability along with the reformulated chance constraint is not a computationally attractive method for obtaining chance-constrained dynamic policies. While it seems theoretically possible to use the NLP formulation of the DC conditions, the encoding would be unwieldy and quickly stretch the limits of current NLP solvers. Therefore, the remaining sections in this chapter are devoted to developing a conflict-directed approach akin to the method introduced by Wang [69] for producing chance-constrained static policies.

For the interested reader, the following two subsections explain in more detail the

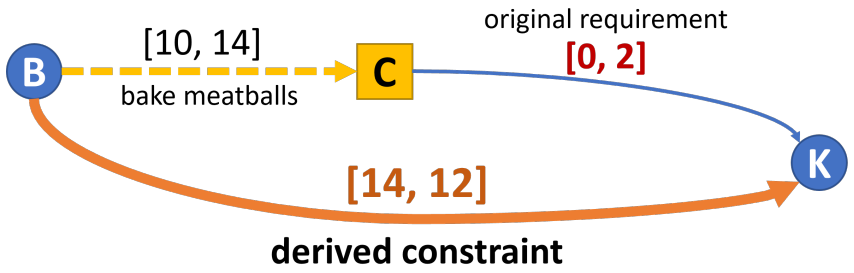

Figure 4-3: When compiling the $C \rightarrow K$ constraint for strong controllability, the C endpoint gets propagated backwards through the $B \rightarrow C$ uncontrollable duration to B. This removes the dependence of the scheduling solution on C's arrival time, which is exactly what strong controllability means. Combining the requirement bounds of $[0, 2]$ with the worst-case outcomes from the duration's $[10, 14]$ bounds, we derive a new constraint $B \rightarrow K$ with bounds of $[14, 12]$. Since the lower bound ends up greater than the upper bound, this is an example of the STNU being *not* strongly controllable.

derivations of the constraints and variables in the full encodings of STNU strong and dynamic controllability.

4.1.1 Strong controllability encoding

I mentioned previously that the semantics of STNU strong controllability are to remove any dependence of the scheduling solution on the outcomes of the uncontrollable durations. Therefore, if a controllable activity or requirement has an *uncontrollable* event as its start or end event, then we must rewrite the constraint so it doesn't involve that event.

Figure 4-3 illustrates this rewriting for the "bake meatballs" activity in our STNU from Figure 2-2. The $C \dashrightarrow K$ requirement begins on the uncontrollable event C, so we must rewrite it. The key insight is that C must be the end event of some unique uncontrollable duration, due to the one-to-one correspondence between uncontrollable events and durations. In this case, that duration would be is $B \dashrightarrow C$. The intuition then is to "propagate" the C endpoint backwards along $B \dashrightarrow C$, resulting in a new derived constraint $B \rightarrow K$. This new constraint only involves controllable events, and it supercedes the original, which we can now ignore.

It remains to discuss how the $[l, u]$ bounds on the new constraint are chosen. In order for the new constraint to guarantee the original's satisfaction in all circumstances, we need to

consider the outcomes of the uncontrollable duration we propagated through. Namely, we consider the extreme cases. If $B \dashrightarrow C$ took the longest duration of 14 to finish, then the *earliest* time we could dispatch K would be immediately after, corresponding to the 0 lower bound of $C \rightarrow K$. Therefore, the *shortest* duration from B to K that we could guarantee *for all outcomes* of $B \dashrightarrow C$ is $14 + 0 = 14$. This becomes the lower bound on $B \rightarrow K$.

Conversely, if $B \dashrightarrow C$ took the shortest duration of 10 to finish, then the *latest* time we could dispatch K would be 2 minutes after. Therefore, the *longest* duration from B to K that we could guarantee is $10 + 2 = 12$, which becomes the upper bound on $B \rightarrow K$.

In this example, we derived a higher lower bound than upper bound on $B \rightarrow K$. Clearly, this is an infeasible constraint. (Or more formally, it's two inequalities that aren't consistent with each other.) Therefore, this segment, and hence the family spaghetti STNU in Figure 2-2, is *not* strongly controllable.

When *encoding* strong controllability though, for our problem of finding a risk allocation, the $[l, u]$ bounds on $B \dashrightarrow C$ would be variables. Thus, we would compile $C \rightarrow K$ into $B \rightarrow K$ with bounds $[u + 0, l + 2]$. Since B and K are controllable events, they would become event variables in the STN encoding, and so this derived constraint would be passed into the solver as the following two linear constraints:

$$\begin{aligned} K - B &\leq l + 2 \\ B - K &\leq -u. \end{aligned} \tag{4.3}$$

This is thus the essence of encoding STNU strong controllability: We get an LP on the event variables and the interval bound variables of interest. There are two more wrinkles to mention to wrap up this discussion.

First, if we have a chain of uncontrollable durations, like $A \dashrightarrow B \dashrightarrow C \dashrightarrow D$, where A is the only controllable event, then propagating an activity/requirement $D \rightarrow E$ back through $C \dashrightarrow D$ simply replaces an uncontrollable event D with another one C. This fix is simple, though: just keep propagating back using the same rules until you hit a controllable event. This is guaranteed, because it is impossible to have a cycle of uncontrollable durations.

Second, in our example above, we had a requirement that *began* on an uncontrollable event C. But we could also have requirements that *end* on an uncontrollable event. Suppose there were such a requirement $D \rightarrow C$ with bounds $[l, u]$. Algebraically, this is equivalent to a requirement $C \rightarrow D$ with inverted bounds $[-u, -l]$. Therefore, we can rewrite all such requirements in this manner, and then the previous reasoning applies.

Originally, strong controllability was formulated in three separate cases [60] depending on whether the start or end event of a given activity/requirement, or both, were uncontrollable. However, one can see that the same reasoning is used for all, and so they can be equivalently mapped to one another.

4.1.2 Dynamic controllability encoding

Like strong controllability, enforcing dynamic controllability can be understood as a set of propagation rules applied to an STNU's activity and requirement constraints. For strong controllability, as per its semantics, the rules allowed us to compile the uncontrollable durations' outcomes out of the picture. For dynamic controllability, we require a weakening of the rules, so that we can observe and react to those outcomes during execution. The form of this weakening is expressed as *conditions* on both the derived constraints and when the rules apply. Hence, rather than having a universal propagation rule, the reasoning gets split into multiple cases, and the constraints derived from that reasoning hold only for certain outcomes during execution.

The key difference to appreciate about dynamic controllability is that the lower and upper bounds of uncontrollable durations have different implications for the scheduling policy. This is due to the inherent forward direction of time during execution, whereby dynamic policies can influence executions in the future, but not those that have already happened. Suppose we had an uncontrollable duration $A \dashrightarrow B$ with bounds $[5, 10]$, and without loss of generality suppose A executes at time $t = 0$.

If there is any event C whose execution time could be impacted by the earliest possible arrival of B at $t = 5$, it is because B's early arrival forces C to be early as well. For example, suppose C must occur before B, but maybe we were lazy with C and still hadn't

executed it by $t = 5$. Then, B might arrive immediately, and it would be too late to fix C. In such cases, then, C must unconditionally assume that outcome for B, and this reasoning is equivalent to how strong controllability handles uncontrollable lower bounds.

Conversely, consider if some event D's execution depended on whether B arrived at its latest time $t = 10$. This would be due to B's late arrival forcing D to execute later as well. So *a priori*, D has to wait for the possibility of B's arrival. However, if B happened to arrive earlier, D could move up in time as well, and a dynamic policy would be able to respond to that. Therefore, any *a priori* propagation on B's arrival affecting D is ***conditioned*** on B arriving at $t = 10$. This is the essential insight for how dynamic controllability differs from strong controllability: Uncontrollable upper bounds get conditionally propagated, and the derived constraints are saved for the dynamic policy to observe and potentially invalidate, as part of their reasoning during execution.

The DC propagation rules – also called *reductions* – formalize this distinction. Since uncontrollable durations' lower and upper bounds are treated differently, and are also distinct from the "ordinary" bounds of the controllable durations and requirements, it helps to separate them into different rules. To aid clarity when expressing these rules, Morris and Muscettola defined the *distance graph* form of an STNU [41]. This is analogous to distance graphs for STNs [17], where each $A \rightarrow B$ constraint with bounds $[l, u]$ gets mapped into an weighted forward edge $A \xrightarrow{u} B$, and an inversely weighted backward edge $B \xrightarrow{-l} A$.

The same principle applies to STNU distance graphs, except that for uncontrollable durations, their $[l, u]$ bounds are inversely mapped and also labeled. Given an uncontrollable duration $A \dashrightarrow B$ constraint with bounds $[l, u]$, its mapped forward edge $A \xrightarrow{b:l} B$ is weighted by the *lower* bound, and the weight has a *lowercase* b label attached. Conversely, the backward edge $B \xrightarrow{B:-u} A$ is weighted by the *upper* bound, labeled by an *uppercase* B. Figure 4-4 demonstrates this on the boiling and straining spaghetti activities of our spaghetti STNU from Figure 2-2.

The interpretation of these edges is as follows: Across all possible duration outcomes for the boiling activity $D \dashrightarrow E$, 8 is the tightest upper bound on $E - D$, but it is conditioned on the outcome being the shortest duration. Similarly, -11 is the tightest upper bound on $D - E$, and it is conditioned on the longest duration outcome. Meanwhile, for

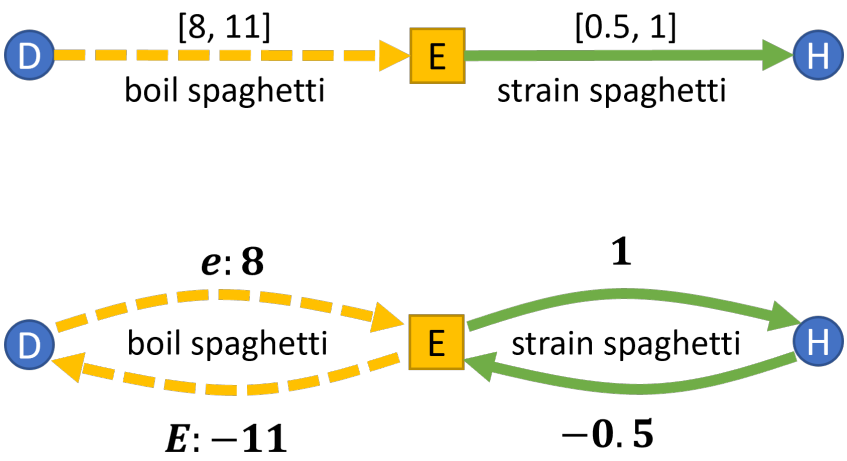

Figure 4-4: In the distance graph form of an STNU, the STN-style $[l, u]$ bounds get mapped to forward edges with weight u and backward edges with weight $-l$. However, for uncontrollable durations, the lower bounds go forward with lowercase labels, and the upper bounds backwards (and negated) with uppercase labels.

the straining activity $E \rightarrow H$, whose duration we control, the $[0.5, 1]$ interval only asks that we spend at least 30 seconds and at no more than 1 minute on it. Therefore, 1 and -0.5 are the only "forward" and "backward" upper bounds, respectively, with no range of outcomes to consider. These distance graph edges, which are not labeled, are called *ordinary* edges. In summary, the distance graph converts all $[l, u]$ duration and requirement bounds into appropriately labeled upper bounds on the algebraic difference between two events' execution times.

Given the distance graph form, we can now concisely express the DC reduction rules. Figure 4-5 illustrates that there are five cases. Each case is based on a triple of event variables and two given distance graph edges. A third edge in red is then derived, with the appropriate weight and label attached.

The first four cases are all variations on a theme: There is a two-edge path $A \xrightarrow{x} B \xrightarrow{-y} C$, from which we derive a new edge $A \xrightarrow{x-y} C$. In all cases, the weight x on the first edge is nonnegative, and the weight $-y$ on the second edge is negative. The derived edge's weight is simply the the sum of those two, $x - y$. This structure has led to these rules being informally called the *plus-minus* reductions [39], and we remind ourselves explicitly of it with the negative sign on the y.

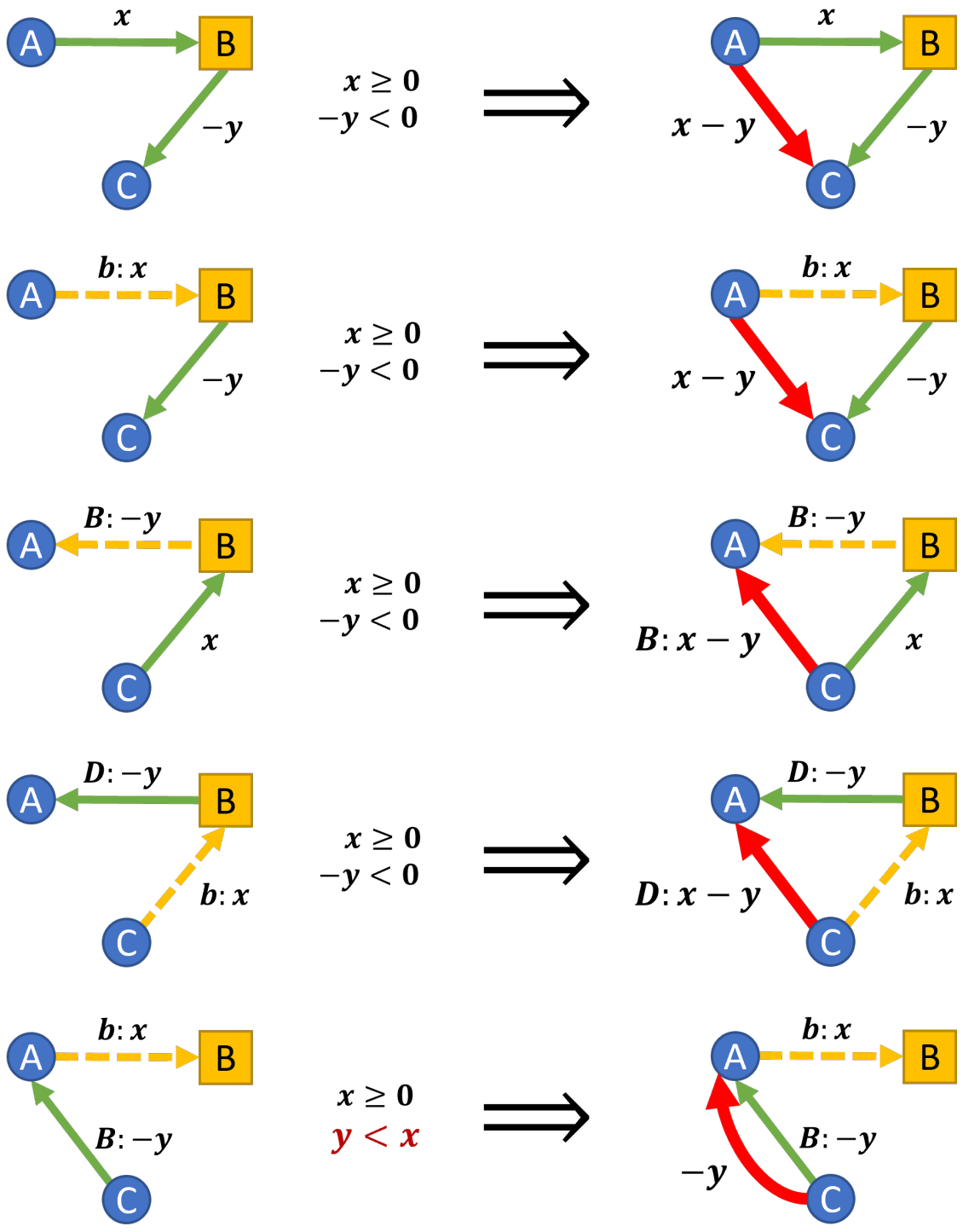

Figure 4-5: The semantics of dynamic controllability imply these reduction rules for deriving additional edges in the STNU's distance graph. Derived edges can be either ordinary or uppercase edges, but not lowercase. The fifth rule turns a derived uppercase edge into an ordinary edge.

What differs among the four cases are the labels on the edges, and their directionality. The first case is called the *ordinary reduction*, because it combines ordinary edges in the same way when propagating in STN distance graphs. Technically, STN propagation isn't usually restricted to plus-minus structure, but there is actually a deep connection STNU DC that justifies this, which we touch on towards the end of this section. The second case is the *lowercase reduction*, and it expresses the idea lowercase edges are unconditionally compiled away in the same way that strong controllability does.

The third and fourth cases involve uppercase edges, and work in the backward direction. The third case is the plain *uppercase reduction*. Here, $B \xrightarrow{B:-y} A$ is the original uppercase edge for the uncontrollable duration $A \dashrightarrow B$. If there is an ordinary edge with nonnegative weight x going into B from C, then we derive $C \to A$ with the uppercase-labeled weight $B : x - y$.

The meaning of this edge is that in any execution scenario where $A \dashrightarrow B$'s duration outcome is y, the difference in execution times between C and A must respect $A - C \leq x - y$. Any valid policy must therefore be aware of this edge *unless* B arrives earlier than y time units after A. This can be rephrased as: Once A has executed, wait at least $x - y$ time units ***or*** until B has executed, before executing C. For this reason, uppercase edges are sometimes called *wait constraints*. These should not be confused with the "wait" actions in our STNU model, nor the `wait` decisions that a policy may output.

Although the third case was illustrated with $B \xrightarrow{B:-y} A$ being an original uppercase edge, it also works with derived edges. That is, suppose the uppercase label was D instead of B. Then $B \xrightarrow{D:-y} A$ must have been an uppercase edge ultimatly derived from an uncontrollable duration $A \dashrightarrow D$. However, the conditional meaning of the uppercase D label still holds, so the reduction rule remains applicable.

This is the logic behind the fourth case, called the *cross-case* reduction. It can be viewed as a combination of the lowercase and uppercase reduction rules. We have an original lowercase edge $C \xrightarrow{b:x} B$ and a derived uppercase edge $B \xrightarrow{D:-y} A$. The end result is the same as the third case, a derived uppercase edge $C \to A$ with uppercase-labeled weight $D : x - y$.

Having shown how uppercase edges are derived, the remaining fifth case addresses

when those uppercase labels can be removed. Note that due to the plus-minus structure, derived uppercase edges always have larger weight (i.e., more positive) than that of the uppercase edge they were derived from. Going back to our example above, where we had an uncontrollable duration $A \dashrightarrow B$ with bounds $[5, 10]$, suppose we eventually derived a new uppercase edge $D \rightarrow A$ with weight $B : -3$. This means we should wait either 3 time units after A or until B has executed before executing D. But the duration model guarantees that B won't arrive until at least 5 time units after A. Therefore, the 3 time limit for waiting will always activate first, and we can treat it just like an ordinary edge. This is what the fifth case expresses.

Technically, the $x \geq 0$ conditions for the second and fifth cases are unneeded. This is because lowercase edges are never derived, and by the STNU model semantics, the original lowercase edges are always nonnegative.

The last point about these rules is why they are conditioned on the plus-minus form. The reason ultimately relates to a deep connection between STNU dynamic controllability and STN dispatchability. An STN is *dispatchable* if enough implied constraints are made explicit in its distance graph, such that when executing it online via a dynamic policy, only local propagations are needed. Local propagation means when an event executes and is assigned a fixed time, that information gets sent to only the event's direct neighbors in the distance graph [43] [58].

Morris showed that to make an STN dispatchable, one simply needs to reduce every nonegative edge $A \xrightarrow{x} B$ in the distance graph against any adjacent negative edge $B \xrightarrow{-y} C$, thus deriving an implied edge $A \xrightarrow{x-y} C$ [40]. This is exactly the form of our four plus-minus STNU reductions. He also observed that an STNU can be viewed as a collection of STNs, where each STN represents a full outcome where the $[l, u]$ bounds on the uncontrollable durations have been collapsed to a single number. These are called the *projections* of the STNU. Therefore, the underlying principle of the STNU reduction rules may be viewed as preserving dispatchability of all the STNU's projections, with the addition of the fifth rule for remove uppercase labels. This is a deep insight that unites the otherwise seemingly complex reduction rules.

Finally, we've covered all the reduction rules of dynamic controllability, but we still

need a final condition to determine whether an STNU is DC. For strong controllability, the test is whether the compiled STN is consistent, which can be determined by encoding the difference constraints with the events as variables. For dynamic controllability, recall that the intuition expressed upfront was that lowercase edges get compiled away, but uppercase edges get propagated and saved for the policy. Without going into all the details, the punchline is that after deriving all possible edges and ignoring the lowercase edges in the distance graph, then for a policy to exist, that distance graph must be consistent in the same sense as an STN's [42] [41].

To summarize, then, the steps to check whether an STNU is dynamically controllable are to: a) derive all possible edges in the distance graph according to the reduction rules, b) throw away the lowercase edges, and c) throw away the *uppercase labels* on the remaining edges, and check for this distance graph's consistency as if it represented an STN. This lends itself to an algorithmic solution for determining dynamic controllabillity, but what we need is a *declarative* set of constraints that fully encode the DC condition, and can be evaluated for truth or falsehood.

Fortunately, we already know how to encode part (c), given our knowledge of encoding STNs from the previous subsection. We just create variables for each event and write down the remaining edges as difference constraints relating them. Note that unlike for strong controllability, here we need to define variables for the uncontrollable events because we keep the original uppercase edges.[1] It is also easy to address (b), by simply not including them in our encoding. What remains is for (a), we need a way to encode all the possible derived edges produced by the DC reduction rules without knowing in advance whether they exist.

The way to solve this issue is to create variables representing the weight for *every* possible edge in the distance graph, derived or not. Then we can assign the variables that to the original, given edges, and let the encoding solve for the remaining derived ones. In our case, for our risk allocation problem, the bounds on the uncontrollable durations are also variables that we are solving for. Thus we leave them unassigned, and if a solution

[1]It may seem paradoxical to create variables for events that will be assigned by Nature, but with the derived uppercase edges included, we are really expressing the idea that we could handle any assignment to them given by Nature.

exists, then we read the risk allocation off of their assignments, while treating all the other variables in our encoding as auxiliary variables.

To this end, we define three sets of edge weight variables, corresponding to the ordinary, lowercase, and uppercase edges, respectively. Ordinary edge weights are denoted w^O_{AB}, representing the weight on distance graph edge $A \xrightarrow{w} B$. Similarly, lowercase edge weights are denoted w^L_{AB}, representing the lowercase-labeled weight on edge $A \xrightarrow{b:w} B$. Finally, uppercase edge weights require parameterization by three events, because the uppercase label can be independent of the edge's endpoints. So to represent the weight w on uppercase edge $A \xrightarrow{D:w} B$, we denote it as $w^U_{AB;D}$.

Recall that the STNU has N events total, K of which are uncontrollable. For ordinary edges, there are no restrictions on which events they may relate, so we need N^2 such variables. Lowercase edges, though are much more restricted. We never derive new lowercase edges, so we only need K of them to represent the original edges. Finally, uppercase edges sharing the same label have the property of always pointing back to the same event. This can be seen in the third and fourth reduction cases, where A is start event of the uncontrollable duration that terminates on B or the D that is not displayed. Since there are exactly K uncontrollable durations, we conclude there are at most K possible events that uppercase edges could end on, and they can start on any of the N events. Thus, we create NK uppercase weight variables.

With these variables, we can encode the DC reduction rules by simply applying the rules to each triple of events, and substituting the edge weight variables into the derivation for each case. Each derivation is conditioned on the plus-minus structure. For example, the uppercase reduction rule on events A, B, and C depicted in Figure 4-5 would be written as

$$\left[(w^O_{CB} \geq 0) \bigwedge (w^U_{BA;D} < 0)\right] \Rightarrow (w^U_{CA;D} \leq w^O_{CB} + w^U_{BA;D}). \tag{4.4}$$

And the label removal rule would be written as

$$(-w^U_{CA;B} \leq w^L_{AB}) \Rightarrow (w^O_{CA} \leq w^U_{CA;B}). \tag{4.5}$$

Note that we ask that the derived edge weight be less than or equal to the sum of the other

two, and not just equal. This is because there might be another triple of events A, E, and C, where either the uppercase or cross-case reduction applies, and we can derive another edge weight value for $w^U_{CA;D}$. We don't want to prevent either reduction from taking place, yet we only have one such variable. We also want to keep the tightest such constraint on the D-labeled uppercase edge weight from C to A. Using the less than or equal inequality lets us achieve this.

Overall, there are $O(N^3)$ triples to consider, and up to 5 rules to apply to each triple, depending on the structure of the STNU. We've shown that the rules, when expressed in terms of our edge weight variables, become logical implications over linear constraints, which was our claim in Equation 4.1. As 5 is a constant, we end up with $O(N^3)$ such constraints in our encoding. Finally, since implications can be rewritten as disjunctions, we thus have our disjunctive linear program as indicated in Figure 4-1, containing N^2+NK+K variables and $O(N^3)$ constraints.

To complete the encoding, we append the linear program for encoding the final consistency of the STNU distance graph with derived edges excluding the lowercase edges. This adds N variables, one for each event, and $N^2 + NK = O(N^2)$ constraints relating the ordinary and uppercase edge weight variables to the event variables.

This concludes our presentation of the encoding for STNU dynamic controllability. As we can see, it is substantially more complex than the encoding for strong controllability. Thus, this thesis develops an alternate algorithmic approach to finding a risk allocation that effectively meets those constraints.

4.2 Conflict-directed hybrid approach

The algorithmic approach we take towards solving our reformulated Problem 3.8 draws ideas from conflict-directed search. We will end up with a *hybrid* solver, because rather than trying to solve the entire problem at once monolithically, we break it down into a master problem and subproblem, and utilize a specialized subsolver to address the latter. Additionally, there is two-way communication between the master and the subproblem.

In our reformulated problem, we are searching within a continuous state space of risk

allocations. However, the principles of conflict-directed search can be illustrated more concretely with discrete variables. Thus, I begin by summarizing its key features in a discrete setting. Then I extend those features to our specific problem.

In a discrete constraint satisfaction problem (CSP), we have a set of variables $\mathcal{X}$ and a set of constraints $\mathcal{M}$ over them [16]. The most straightforward way to solve this problem is to traverse a search tree that branches on the domain values for each variable, and to check constraint satisfaction at the leaves. The complexity of this is clearly exponential in the number of variables, which is the search tree's depth.

The key feature of conflict-directed search is to separate out a subproblem determined by the assignment to a subset of variables $\mathcal{Y}$, sometimes called the *decision* variables [72]. The subproblem is defined over the remaining variables $\mathcal{Z} = \mathcal{X} \setminus \mathcal{Y}$. One advantage of this is that the search tree can stop branching after all decision variables have been assigned, thus reducing its depth.

The real advantage, though, is the ability to use a specialized solver for the subproblem If that solver can return a *conflict* when the subproblem is infeasible, then the top-level search can use that conflict to inform subsequent candidate assignments to the decision variables. To understand what a conflict means and how that works, we need to characterize the relationship between the top-level problem of assigning $\mathcal{Y}$, often called the *master* problem, and the subproblem. Figure 4-6 depicts this decomposition of the original CSP.

When we partition the original set of variables $\mathcal{X}$ into $\mathcal{Y}$ and $\mathcal{Z}$, it also partitions the constraints as follows. Some constraints $\mathcal{A}$ are defined only over variables in $\mathcal{Y}$, depicted in orange. Others in $\mathcal{B}$ are only over subsets of $\mathcal{Z}$, depicted in green. Finally, there are those in $\mathcal{C}$ which involve variables from both $\mathcal{Y}$ and $\mathcal{Z}$.

Thus, when the master searches over $\mathcal{Y}$, it is responsible for checking whether the constraints in $\mathcal{A}$ are satisfied. If so, then the subproblem is responsible for assigning variables in $\mathcal{Z}$, and all the constraints in $\mathcal{C}$ fall under its purview. However, it must also make sure that its assignments to $\mathcal{Z}$, combined with the master's assignment to $\mathcal{Y}$, satisfies $\mathcal{B}$.

For example, suppose the purple constraint in Figure 4-6 requires y_3, z_1, and z_3 to all be different, and suppose all the variables have the domain $\{\texttt{red}, \texttt{green}, \texttt{blue}\}$. Then if the master assigns $y_3 = \texttt{red}$, that removes `red` from z_1's and z_3's domains. That means

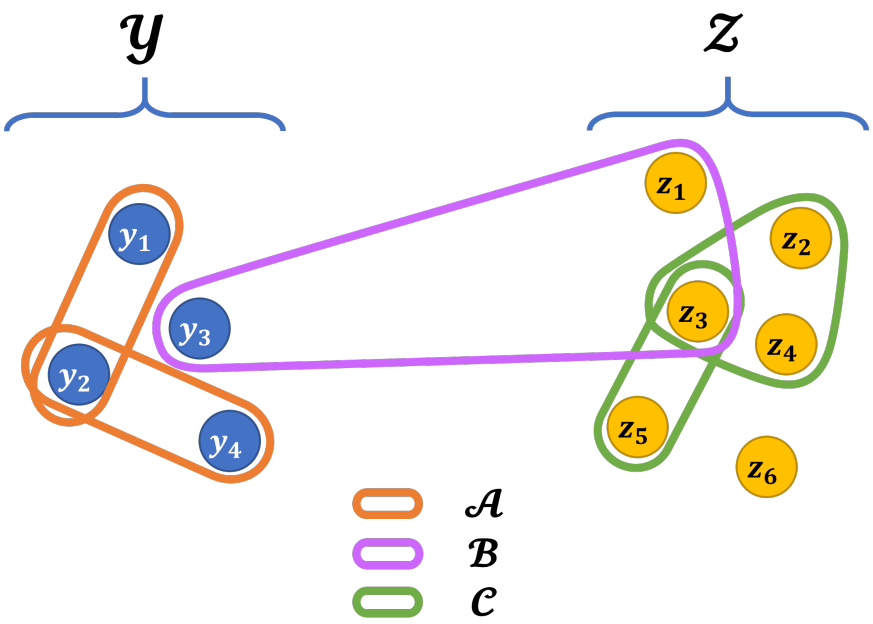

Figure 4-6: A CSP is a set of constraints defined over a set of variables, which we partition into $\mathcal{Y}$ and $\mathcal{Z}$. Each constraint is defined over a subset of the variables $\mathcal{X}$, depicted as a covering over that subset. The colors of the constraints indicate whether they involve only variables in $\mathcal{Y}$, only variables in $\mathcal{Z}$, or variables from both.

the subproblem has to consider the *projected* constraint that (z_1, z_3) can only be assigned $(\texttt{green}, \texttt{blue})$ or $(\texttt{blue}, \texttt{green})$.

Therefore, while the master problem comprises constraints $\mathcal{A}$ just on the decision variables, the subproblem structure includes constraints $\mathcal{C}$ on the non-decision variables, *plus* the constraints $\mathcal{B}$ *conditioned* on the assignments to $\mathcal{Y}$ and *projected* onto the space of $\mathcal{Z}$. We call these projected constraints $\mathcal{B}^{\mathcal{Y}}$. Given this structure, we can say that if the master successfully assigns $\mathcal{Y}$ satisfying all of $\mathcal{A}$, followed by the subproblem successfully assigning $\mathcal{Z}$ satisfying all of $\mathcal{B}^{\mathcal{Y}}$ and $\mathcal{C}$, then we have a full solution to the original CSP.

What's interesting, though, is if the subproblem turns out to be infeasible. In that case, we would like the subsolver to return a *conflict*, which can be viewed as a subset of the subproblem input $\mathcal{B}^{\mathcal{Y}}$ and $\mathcal{C}$. We claimed above that such a subset could help inform the master in searching for the next candidate $\mathcal{Y}$ assignment, and here's how. First of all, if the conflict only involves constraints in $\mathcal{C}$, then the original CSP is always infeasible. No further assignments to $\mathcal{Y}$ can help remove that conflict.

But if the conflict involves projected constraints in $\mathcal{B}^{\mathcal{Y}}$, we can implicate the subset of

assignments to $\mathcal{Y}$ those constraints were conditioned on. Back to the example with the purple constraint in $\mathcal{B}$, once y_3 got assigned `red`, it became a constraint in $\mathcal{B}^{\mathcal{Y}}$, defined just over z_1 and z_3. Call it $b_1^{\mathcal{Y}}$. Now consider the green constraint in $\mathcal{C}$ on z_3 and z_5, and call it c_1. Suppose c_1 only allowed combinations where $z_3 = \texttt{red}$. Then $b_1^{\mathcal{Y}}$ and c_1 would form a subproblem conflict.

From the master problem's point of view, there would be no value in ever framing another subproblem with the same projected constraint $b_1^{\mathcal{Y}}$. It would just guarantee that the same conflict appears in the subproblem, rendering it infeasible. So the master *learns* that it should avoid assigning $y_3 = \texttt{red}$. This thus reduces the branching factor on y_3's nodes from 3 to 2, and this is the main feature of conflict-directed search: It lifts subproblem conflicts into new constraints for the master, called *conflict resolutions*, which enable it to prune large swaths of the remaining search space.

The idea of using conflicts to guide discrete search has a rich history [53] [14] [26], with conflicts going by different names, such as *nogoods* or *elimination sets*. It is a versatile concept that can be applied in many settings. For instance, Williams and Ragno [72] combined conflict-resolution techniques with informed A* search to efficiently solve CSPs with an objective function. And satisfiability modulo theories (SMT) solvers often rely on the subtheory solvers to return conflicts that further guide the master-level boolean SAT solver [21] [15].

SMT solvers exemplify an important aspect of using conflicts, because some subtheories may involve constraints on *continuous* variables, while the master SAT solver is still operating on discrete values. This means conflicts have to be translated from a continuous space into a discrete space. In many planning scenarios, one may have to make discrete choices while accounting for continuous state, such as spatial localization, scheduling decisions, or even probabilistic belief states. Similar translations of conflicts from continous spaces have thus been used [52] [56] to inform search over the discrete choices.

With the above summary of the structure of conflict-directed search, we turn our attention now to applying its principles to our reformulated risk allocation problem. The key is to find an appropriate partition of the variables and constraints so as to separate out a master problem and subproblem. What's notable here is that we are searching within an entirely

continuous space of risk allocation variables. The previous section also noted that encoding STNU controllability required event variables, and in the case of dynamic controllability, additional edge weight variables. Thus, there are no discrete variables at all, so assigning variables in the master problem will need to involve a method other than discrete search.

Fortunately, the structure of our reformulated problem suggests a very natural partitioning, along with the appropriate solving methods. In Chapter 3, we had argued for finding a risk allocation that yields a controllable STNU. Figures 3-8 and 4-1 then illustrated the clear distinction between satisfying the reformulated chance constraint, and then making sure the implied STNU is controllable. Thus, the key insight is to define our master problem as finding a risk allocation, and then the subproblem as checking STNU controllability.

This avoids some of the difficulties, discussed in Section 4.1, of solving at once the nonlinear reformulated chance constraint together with the full encoding of STNU controllability. Namely, we could first use an NLP solver to assign the risk allocation variables while satisfying the nonlinear reformulated chance constraint. Then, we could employ more specialized solvers just for the controllability conditions. For strong controllability, an LP solver would suffice, whereas for dynamic controllability an MLLP or MILP solver would be appropriate. If the subsolver determines the STNU is controllable, then we have a solution, and any policy for the STNU will satisfy our original cc-pSTN problem.

The resulting algorithm flow is depicted in Figure 4-7. Note that when the master generates a risk allocation, this implies a grounded STNU, where all the $[l, u]$ bounds become constants. Thus for the remaining constraints expressing controllability, we would turn all the l's and u's into their assigned values. This corresponds to the "conditioning" and "projection" steps in the conflict-directed search paradigm.

This picture is missing, though, the key concept of conflict extraction from the subsolver. There is no guarantee that the first risk allocation generated by the master problem will result in a controllable STNU. Therefore, we need the subsolver to return a conflict if it's not controllable. The master records the conflict, so that when generating the next and any subsequent risk allocations, it can explicitly avoid creating a subproblem with that same conflict. Figure 4-8 updates the diagram to indicate this.

This raises two questions for applying the conflict-directed paradigm: First, we need

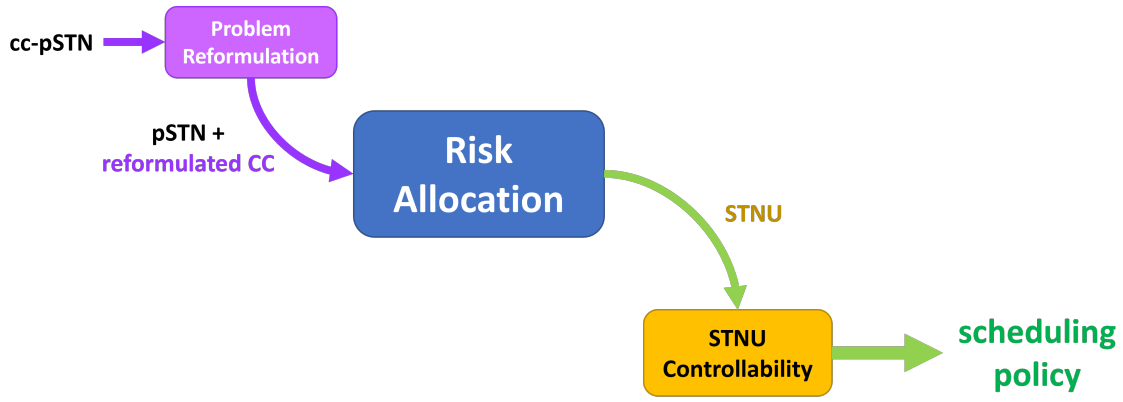

Figure 4-7: In the master-subproblem decomposition of our reformulated problem, we first solve for the risk allocation, and then check controllability of the implied STNU.

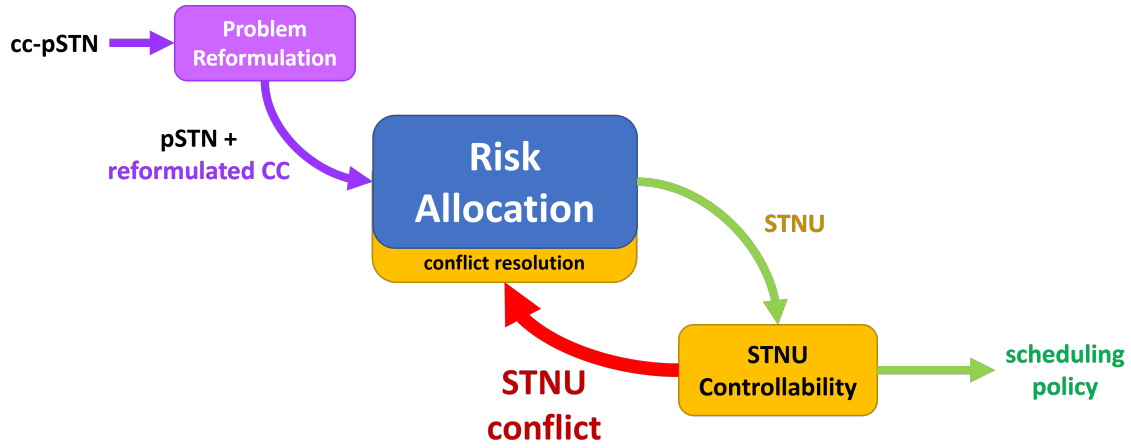

Figure 4-8: When the STNU implied by the risk allocation is not controllable, the master problem needs a conflict returned back, so it can make sure the subsequent risk allocation resolves that conflict.

to figure out how the subsolver will detect and extract such conflicts. Second, we need to understand how to map those conflicts back into the master problem's space

To answer the first question, it helps to step back and consider what the subproblem is asking, separate from how the subsolver achieves it. All we need to know is whether a given STNU is controllable, whether strongly or dynamically, and if not, we should return a conflict that is a subset of the STNU's activities and requirements. So far, we've been envisioning that we encode the controllability conditions as constraints, and then feed them to a black box LP or MLLP/MILP solver. Unfortunately, it is difficult for such black boxes to return conflicts. While other have studied methods to extract *inreducible infeasible subsets* of the input constraints [11] [27], typically these involve running different combinations of constraints through the solver again. The result is a rather inefficient, general-purpose mechanism, if the solver supports it at all.

The key insight is to recognize that we have specialized algorithms for checking STNU controllability, both in the strong [60] and dynamic cases [33] [38] [39]. Furthermore, these algorithms rely on a fundamental property of STNs and STNUs that equates infeasibility with a very specific form of conflict, namely a *negative cycle* in their distance graphs. Section 4.3 will detail the exact nature of these conflicts. For now, it suffices to know that these algorithms operate on the distance graphs to explicitly search for such conflicts. When a conflict found, it becomes trivial to return them as proof of uncontrollability.

These algorithms are also quite efficient, performing in $O(NM)$ for strong controllability, and $O(N^3)$ in the dynamic case. These runtimes are inherited from classical shortest-path graph algorithms that the methods rely on. Note that these *time* complexities closely match the *space* complexities of simply writing down the input constraints of the full controllability encodings. As general-purpose LP and MILP solvers operate beyond polynomial time, we can expect significant savings in using the specialized STNU algorithms.

Therefore, in Figure 4-8, we can swap out the subproblem's black box solver for a specialized STNU controllability checker. This leads to the second question, which is how to translate STNU conflicts into the risk allocation space for the master. Since this depends on detailed knowledge of those conflicts' structure, I defer the full discussion to Section 4.3.

However, I can make the following observations now.

Each edge in an STNU's distance graph corresponds to either an l bound or an u bound between two events, whether it comes from an activity or a requirement. A *cycle* in the distance graph, then, is a collection of those weighted edges. The central point in the STNU conflict is that the sum total of those edges' weights is negative. Therefore, the *resolution* constraint for a conflict is to make that sum total non-negative.

In the fully grounded STNU that the subproblem receives, all the edges have constant weight. But those weights can easily be traced back to whether they were actually constants in the pSTN, corresponding to $[l, u]$ bounds on *controllable* durations and requirements, or whether they were the assignments to some risk allocation variables, corresponding to $[l, u]$ bounds being assumed on *probabilistic* durations.

Thus, when given an STNU conflict, we can compose an arithmetic expression stating the fact that a series of constant edge weights adds up to less than zero. Then we replace certain terms of that addition with their corresponding l or u risk allocation variables. In this manner, we "lift" the conflict from the grounded STNU into an expression in the risk allocation space. Finally, we *resolve* that conflict by negating the expression (typically by reversing the inequality), and that forms a new constraint for the master risk allocation problem. This is the meaning of the yellow "conflict resolution" that gets attached to the risk allocation block in Figure 4-8.

It is worth noting that by using STNU controllability-checking algorithms to extract conflicts, we avoid having to introduce new variables for the events, and in the case of dynamic controllability, variables for the derived edge weights. Not only does this speed up the subsolver, it also simplifies the form of the conflicts and the task of translating them for the master. If we had fed the full encoding to a black box for the subsolver, the irreducible infeasible subsets we'd get as conflicts would contain constraints relating all those extra variables, and many of those constraints would not be part of the input STNU. Instead, the negative cycle form that our approach uses neatly summarizes how all those "internal" variables and constraints would have composed to point to the relevant portions of the STNU.

It remains to discuss the termination conditions of this conflict-directed solving ap-

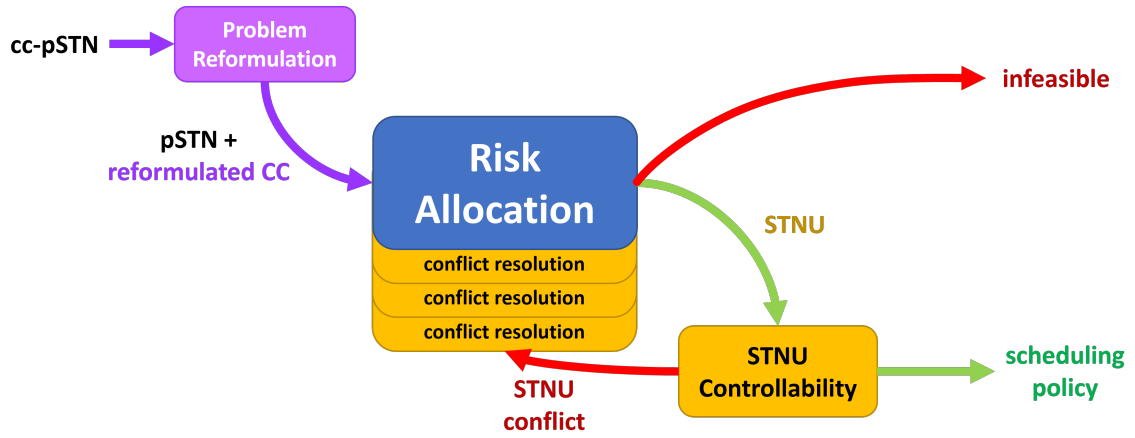

Figure 4-9: After multiple rounds of risk allocation and collecting conflicts, either the STNU will be controllable, or the collected conflict resolutions will be incompatible with the reformulated chance constraint, leading to infeasibility in the master solver. These are the only two ways our conflict-directed algorithm will terminate.

proach. Figure 4-8 introduces a loop into the algorithmic flow, so the question is when do we break out of the loop with an answer. We already mentioned, using Figure 4-8, that when we find a risk allocation that yields a controllable STNU, we have an affirmative answer and may return any policy of this STNU. This is what the green arrow in the bottom right represents. This may not happen on the first round, but throughout subsequent rounds of conflict discovery, we can take that exit as soon as controllability is satisfied.

Alternatively, the master problem might eventually collect enough conflicts that it becomes infeasible to resolve them along with satisfying the reformulated chance constraint. Since these conflicts must be resolved in *any* instance of the subproblem, this means the master is unconditionally infeasible[2], and hence so is the entire reformulated problem. In this situation, we exit out of the master, returning infeasible. This is shown in Figure 4-9.

Although we have identified the ways in which this algorithm would terminate, we haven't yet proven that it would *actually* terminate. That argument rests on the claim that

[2]One could also ask whether the subproblem could be unconditionally infeasible, and what that means. The subproblem is inherently conditioned on the assignments the master makes for the "decision variables", in this case, the risk allocation. Recall that our subproblem conflicts are negative cycles in the STNU distance graph. And the cycle weight is conditioned on the risk allocation, which is why we translate the cycle weight expression back in terms of the risk allocation variables. But if *none* of the edges in the cycle mapped back to a risk allocation variable (i.e., a bound on a probabilistic duration), then the cycle weight is *unconditionally* negative in all instances of the STNU controllability subproblem. In this case, we could either return infeasible from the subproblem directly, or pass that unconditional conflict back up, and let the master figure out that the conflict resolution would be self-inconsistent.

there are only a finite number of possible conflict resolutions we could attach to the master problem. Again, a full discussion relies on understanding the structure of these conflicts and their resolutions, which is presented in Section 4.3. Therefore, I defer the termination guarantee to Section 4.4.

Here is the gist of the argument, though: Within a given STNU structure, there are a finite number of negative cycle topologies we could discover proving its uncontrollability. So if we keep going around the loop, any new STNU conflicts would map into existing conflict resolutions, leaving the master problem unchanged. Hence, in theory, once we've exhausted those possibilities, the algorithm would be forced to satisfy one of the two exit conditions.

In practice, it is highly unlikely that we would reach this situation. Our experiments in Chapter 7 consistently require fewer than 10 iterations to exit the loop. And often, approaching 10 iterations is a sign that the problem is tightly constrained, and that the master solver is likely to fail.

The conflict-directed algorithmic approach I've presented for solving our reformulated problem is formalized as pseudocode in Algorithm 4.1. Lines 7 and 11 correspond to solving the master problem and the subproblem, respectively. To satisfy the original cc-pSTN problem specification, when the subproblem is controllable, we generate and return any feasible policy for the STNU on line 13. Implicit is that choosing whether to check strong or dynamic controllability on line 11 determines whether we produce a static or a dynamic policy.

In practice, the decisions of an STNU policy can be generated online during execution. For dynamic policies, Hunsberger [33] has formally stated such an algorithm, based on the principles outlined by Morris [38]. In the case of static policies, since STNU strong controllability reduces to the compiled STN over its controllable events, we just need a full schedule for the STN. However, STNs can also be dispatched online, via an extra preprocessing step that puts them in dispatchable form [17]. Muscettola, Tsamardinos, and Morris [43] [58] [40] have given algorithms for generating efficient dispatchable forms, as well as theoretical characterizations of them.

One aspect of Algorithm 4.1 not yet discussed is the ability to return a conflict *for the cc-*

Algorithm 4.1: Conflict-directed hybrid solver for the reformulated cc-pSTN problem

Input: A pSTN $\mathcal{N}^p$
Input: A chance constraint with risk bound Δ
Output: Whether a scheduling policy for $\mathcal{N}^p$ with failure probability $\leq \Delta$ was found
Output: If yes, the policy
Output: If no, a conflicting subset of the cc-pSTN

```
// Reformulate the chance constraint and initialize the master problem
ra-vars ⟵ Instantiate risk allocation variables
reformulated-cc ⟵ Express reformulated chance constraint
master-problem ⟵ Instantiate master problem object
AddVariables(master-problem, ra-vars)
AddConstraint(master-problem, reformulated-cc)

while true do
    // Solve the master problem; return if infeasible
    feasible, ra-assignment, infeasible-subset ⟵ RunSolver(master-problem)
    if not feasible then
        return false, MapToCCpSTN(infeasible-subset)

    // Solve the subproblem; return if controllable
    stnu ⟵ GenerateSTNU(N^p, ra-assignment)
    controllable, negative-cycle ⟵ CheckControllability(stnu)
    if controllable then
        return true, GeneratePolicy(stnu)

    // Translate STNU conflict into master problem
    conflict ⟵ TranslateIntoRASpace(negative-cycle)
    conflict-resolution ⟵ NegateConstraint(conflict)
    AddConstraint(master-problem, conflict-resolution)
```

pSTN itself when it cannot find a chance constraint-satisfying policy. Technically, this was not specified in the original cc-pSTN Problem 2.20. However, this is a useful capability to consider when integrating with planners. On page 123, I had mentioned that conflicts from continous spaces can help planners guide their search over discrete choices. Thus, when checkers for STN consistency and STNU controllability return negative cycles, whoever uses those checkers as a subroutine can translate those cycles into learned constraints, just as Algorithm 4.1 does for risk allocation, but for a discrete space. Likewise, it would be useful for our cc-pSTN solving algorithm to return some form of conflict when infeasible.

With respect to a problem's inputs, a conflict is basically a subset of them. The smaller the subset, the more powerful the conflict is, because the conflict resolution then prunes a larger portion of the space of possible subproblems. In our solution approach, infeasibility is determined by the master problem becoming overconstrained. Thus, we rely on the master solver to identify a subset of the learned conflict resolutions that are incompatible, either with themselves or with the reformulated chance constraint. Ideally, it would be an *irreducible* infeasible subset (IIS), which makes the conflict as small as possible. This subset is precisely what's responsible for us returning *false* on line 9, so we map those conflict resolutions (plus the reformulated chance constraint if included) back into the original cc-pSTN's components. *That* is what we return as our cc-pSTN conflict.

The weak link in this strategy, though, is our reliance on a third-party solver to return such an IIS. Not all solvers support this feature, especially when a nonlinear constraint is involved. In that case, one would have to implement a custom strategy that wraps around the solver. Doing so would be a problem in and of itself, so I have chosen to descope it. At the very least, one could always treat the entire set of conflict resolutions plus the chance constraint as the infeasible set. It just means the conflicts returned are not miminal and thus less informative.

To fully implement Algorithm 4.1, we need to understand the form of the negative cycles returned by the controllability checkers, as well as how lines 14 and 15 process it as a conflict to be learned by the master problem. In the following Section 4.3, I will discuss these in turn for strong and dynamic controllability. Specifically, I will focus on what the conflict resolution constraints look like in the risk allocation space, and what that means for the

master solver. Finally, Section 4.4 will discuss termination and soundness/incompleteness guarantees.

4.3 Extracting and resolving conflicts

The previous section claimed that the form of conflicts returned by the subproblem were negative cycles in the STNU distance graph. The purpose of this section is to illustrate why that is the case, and then given those cycles, what the master problem needs to do to resolve them in subsequent risk allocations. That is, we will be translating the key condition(s) of those cycles into a constraint in the space of risk allocation. The constraint's negation then becomes our desired conflict resolution constraint, which gets fed into the master's solver.

The form of the conflict resolutions has important implications for the choice of solver used by the master problem. Namely, the solver must be capable of processing those constraints. What we will see is that strong controllability conflicts result in purely linear conflict resolution constraints, whereas dynamic controllability conflicts require *disjunctive* linear resolutions. Correspondingly then, when searching for a static or a dynamic policy, the conflict resolutions form either a linear program or a *mixed-logic* linear program.

You will note that these are exactly the same class of constraints needed for the full encodings of strong and dynamic controllability, given previously in Section 4.1. Therefore, we haven't changed the type of solver required to solve the reformulated chance-constrained problem, going from the full encoding to the conflict-directed approach. This is perhaps no accident, as we can view the conflicts as an incremental buildup of an alternate encoding for strong or dynamic controllability.

What we *have* gained, though, is precisely this incremental buildup. I mentioned previously that Wang [69] demonstrated significant runtime savings using the conflict-directed approach to produce static policies. This can be attributed to avoiding the full encoding's $O(N)$ additional variables completely, and its $O(N^2)$ additional constraints upfront. Also, the full encoding's constraints are each limited to relating just two events in the temporal network, whereas a negative cycle has the potential to summarize how multiple activities' and requirments' bounds compose into a conflict. These cycles can thus

be viewed as being more "informative" than just single edge constraints, and this contributes to the conflict-directed approach needing relatively few iterations.

We would expect these gains to carry over at least as much for dynamic controllability conflicts. We still get the expressive power of negative cycles, *and* we are now discovering them in lieu of encoding $O(N^2)$ additional variables and $O(N^3)$ disjunctive linear constraints.

The fact remains, though, of having to solve either an LP or MLLP in conjunction with the nonlinear reformulated chance constraint. Like with the full encoding, an NLP solver suffices for generating static policies, but more sophisticated methods are needed to handle the dynamic case. One might consider alternative MILP or NLP formulations of the MLLP, others had provided for the DC full encoding, described in Section 4.1. However, those formulations were specific to the full encoding, and thus not directly applicable to the DC conflict resolutions we seek here.

Therefore, simply finding a black box solver to substitute in line 7 of Algorithm 4.1 will not suffice for generating dynamic policies. Nevertheless, the general spirit is still valid towards approaching a final solution. The focus of this chapter, then, is to derive the form of the DC conflict resolutions, and Chapter 5 will provide additional algorithmic structure to process those resolutions, resulting in a complete algorithm. Because DC conflicts build on concepts of SC conflicts, I present the latter first in the next subsection. It is also the case that since all strongly controllable STNUs are dynamically controllable, SC conflicts can be viewed as a special case of DC conflicts.

4.3.1 Strong controllability conflicts

I begin with a summary of how we process and resolve strong controllability conflicts. The remainder of this subsection then illustrates that process on our favorite spaghetti example.

In Subsection 4.1.1, I demonstrated that strong controllability is enforced by compiling away constraints' dependence on the arrival times of uncontrollable events. The intuition was to push any activities' or requirements' endpoints back along the uncontrollable durations, and absorb the uncertain time window into the original constraint's controllable slack.

This is exactly what strong controllability-checking algorithms do [60], thus mapping the STNU down into an STN on the controllable events.

The STN's consistency is thus equivalent to the STNU's strong controllability. By the Fundamental Theorem of STNs [17] [33], an STN is consistent if and only it has no negative cycles in its *distance graph*. Therefore, we run standard single-source shortest paths (SSSP) algorithms [10] on the distance graph. If a negative cycle exists, these algorithms are guaranteed to return a *simple* such cycle (i.e., no repeated vertices). If a negative cycle is returned, we can trace each edge of that cycle back to the original STNU's interval bounds, and this will form a (potentially non-simple) negative cycle in the *STNU's* distance graph.

STNU controllability is well-defined only for fully grounded STNUs, where the intervals bounds are given as constants. Therefore, this is the input to any controllability-checking algorithm. Internally, this gets translated into a distance graph with the constant bounds as edge weights, so any negative cycle returned as a conflict will be in terms of those weighted edges.

However, our master problem needs to know how that conflict helps prune the risk allocation space. That is, we need to derive a constraint that helps it avoid, or *resolve* that conflict for subsequent risk allocations. The first step, then, is to express the current fact that our candidate risk allocation yielded that negative cycle. Once we have that, we can simply negate the expression to enforce that no further risk allocation may produce that negatice cycle again. This is what lines 14 and 15 in Algorithm 4.1 are meant to achieve.

Now the STNU remains topologically unchanged between different instances of the subproblem, because it is always based on the original pSTN's structure. Therefore, that cycle will always exist; it's just a question of whether its weight is negative. So the expression we want is that the cycle's weight is currently negative.

$$\sum_{\text{edge}\in\text{cycle}} weight(\text{edge}) < 0. \tag{4.6}$$

For the STNU that was passed to the subsolver, all the weights on the left-hand side are constants, and this is a true statement. However, we want it to be a statement about what we want all such STNUs to avoid in the future. So for any edge that corresponds to a risk

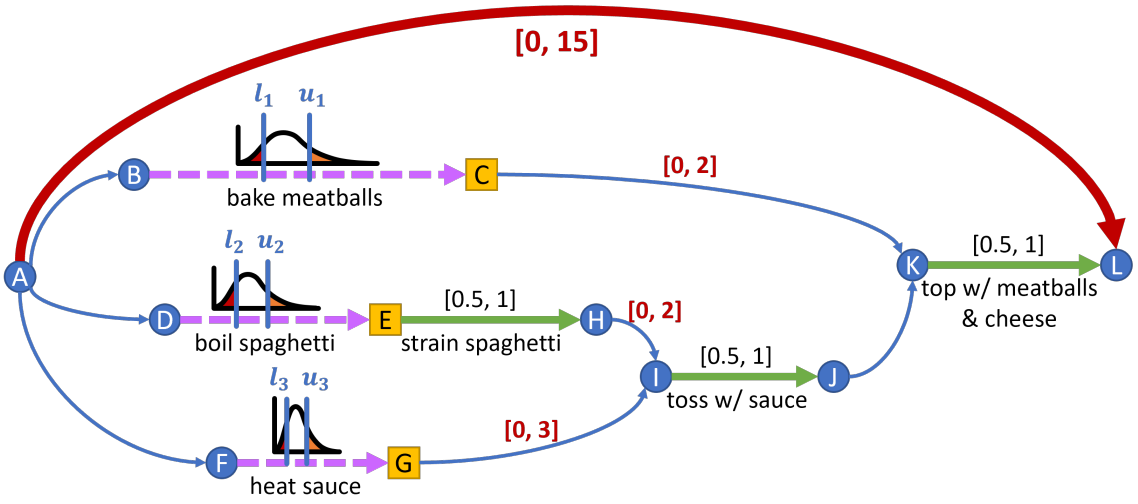

Figure 4-10: This is the original pSTN with the risk allocation variables about to be imposed on its uncontrollable durations.

allocation bound on an *uncontrollable duration*, we replace its assigned weight with the risk allocation variable itself. What we'll have then, is an inequality in terms of the risk allocation variables This basically identifies a region of the risk allocation space where that cycle would have negative weight. So by flipping the inequality into a greater than or equal, we create a constraint that tells the master problem to avoid that region.

Now we apply this process to the spaghetti scenario from Chapter 2. Figure 4-10 the pSTN with risk allocation variables hovering over its uncontrollable durations. (This is a repeat of Figure 3-3.) Once we assign those variables, line 10 of Algorithm 4.1 formulates a grounded STNU out of that, and this becomes the subproblem. Figure 4-11 depicts that STNU for the risk allocation assignment $(l_1, u_1) = (10, 14)$, $(l_2, u_2) = (8, 11)$, and $(l_3, u_3) = (2, 4)$.

Inside the function that checks strong controllability, new constraints are derived according to the principles in Subsection 4.1.1. For example, the C endpoint on the $[0, 2]$ requirement of $C \rightarrow K$ gets compiled back through the "bake meatballs" uncontrollable durations $B \dashrightarrow C$. According to the rule, this creates a new "virtual requirement" on $B \rightarrow K$ with lower bound $u_1 + 0$, and upper bound $l_1 + 2$. This is shown in red in Figure 4-12. The same rule is applied to the $E \rightarrow H$ controllable activity and the $G \rightarrow I$ requirement.

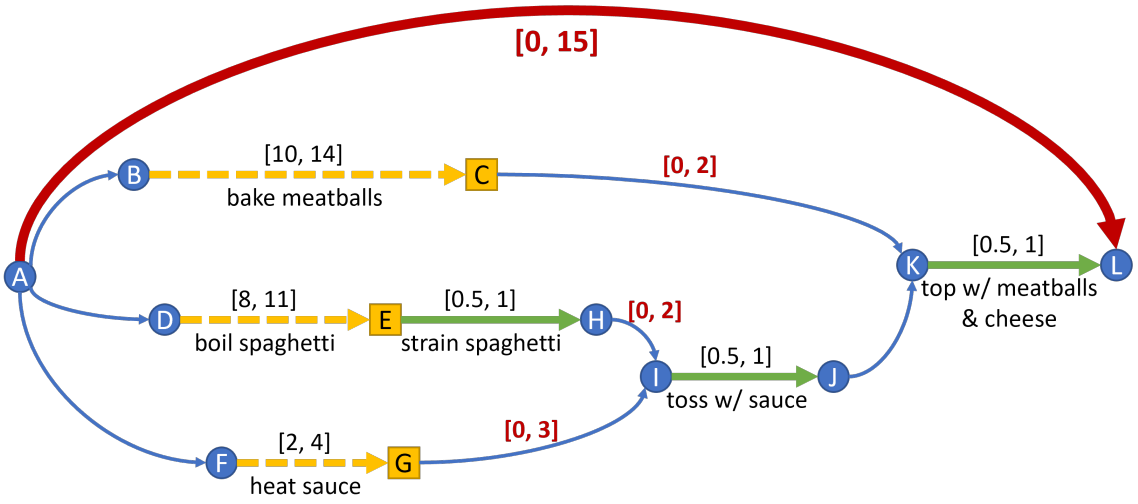

Figure 4-11: Once the risk allocation variables are assigned, a fully grounded STNU is posed as the subproblem.

When we bring in the assigned values of the risk allocation variables, the SC derivation rules thus produce a $[14, 12]$ requirement on $B \rightarrow K$, an $[11.5, 9]$ controllable duration on $D \rightarrow H$, and a requirement $[4, 5]$ on $F \rightarrow I$. These new constraints *dominate* the original $C \rightarrow K$, $E \rightarrow H$, and $G \rightarrow I$ constraints, respectively, in that if we satisfy the new ones, the original ones are guaranteed to be satisfied, too, for any outcome of the uncontrollable durations. This means we can safely ignore any old constraints that touched an uncontrollable event. We can also ignore the uncontrollable durations themselves, because we have compiled away any dependence on their outcomes. This is shown in Figure 4-13.

Since the remaining constraints only connect the controllable events, what we have is an STN. This thus illustrates how STNU strong controllability is reduced to STN consistency. If we can find a static schedule to the STN's events, then a valid static policy for the STNU can be built from it. Now, in this particular STN, one can intuitively see that the derived $B \rightarrow K$ and $G \rightarrow H$ constraints are impossible to satisfy, because their lower bounds are greater than their upper bounds. The remaining discussion will show how these inconsistencies are detected formally, and thus what is the structure of the negative cycle conflict we find.

I claimed earlier that an STN's consistency can be determined by checking for negative

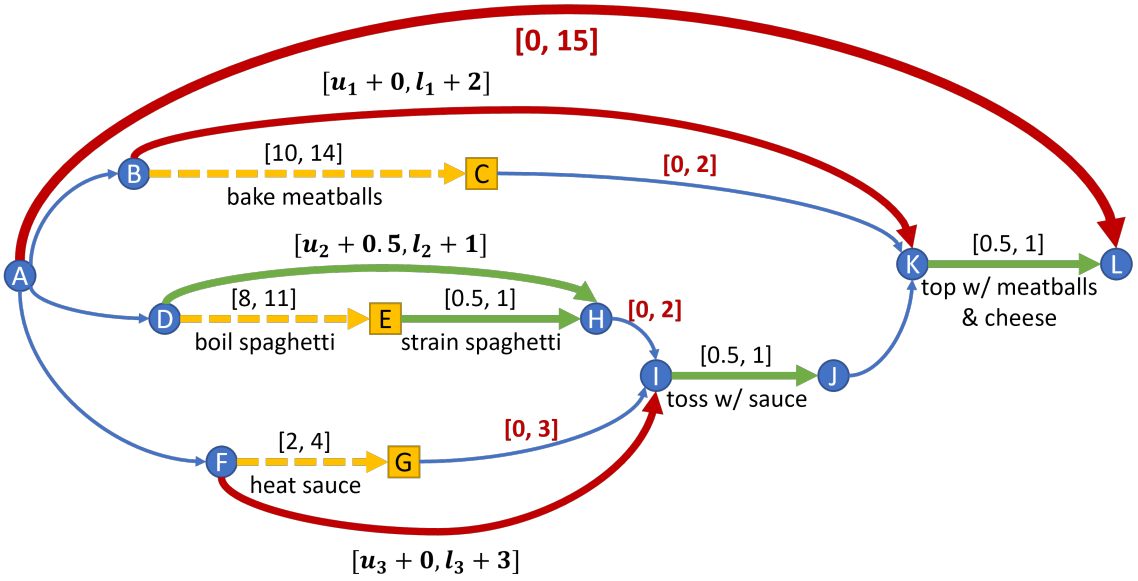

Figure 4-12: To enforce strong controllability, any controllable activities or requirements that touch uncontrollable events are compiled according to the application of the rules shown.

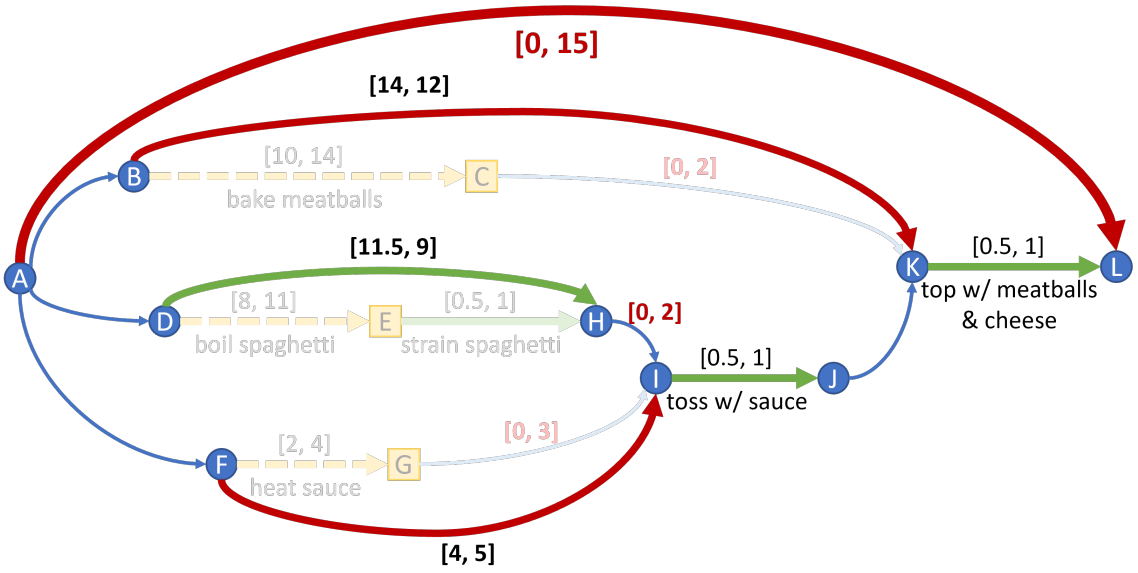

Figure 4-13: This is the result of applying the strong controllability compilation (or reduction) rules. What's left is an STN on the controllable events, and we can ignore any constraints that touch an uncontrollable event.

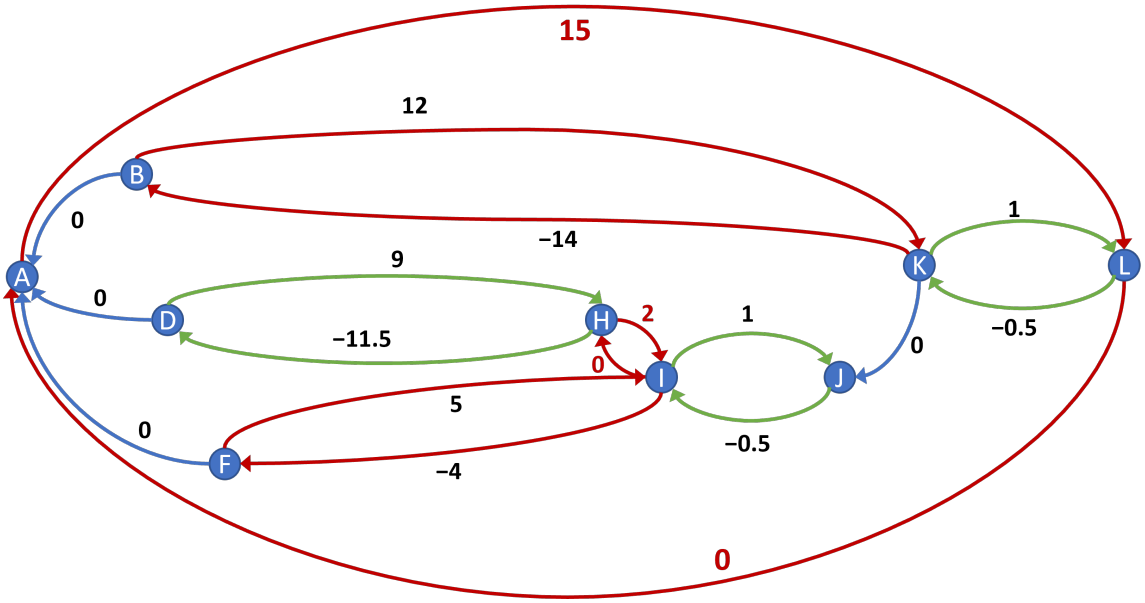

Figure 4-14: This is the distance graph of the STN. Upper bounds become forward weighted edges, and lower bounds become reverse negatively-weighted edges.

cycles in its *distance graph*. The idea behind constructing this graph is that each constraint in an STN really imposes two distinct difference inequalities. This is true regardless of whether a constraint represents an activity (including wait activities) or a requirement. Earlier in Subsection 4.1.1, we saw this idea at work through Equation 4.3.

In an STN's distance graph, all the events become vertices, but every constraint $A \to B$ with bounds $[l, u]$ becomes two weighted edges: a forward edge $A \xrightarrow{u} B$ and a backward edge $B \xrightarrow{-l} A$. Figure 4-14 is thus the distance graph of the STN in Figure 4-13. Remember that the wait activities $A \to B$, $A \to D$, $A \to F$, and $J \to K$ all have interval constraints $[0, +\infty)$. An upper bound of $+\infty$ is really no constraint at all, so those edges are missing in the distance graph, but the backward edges with 0 weight remain. Also, the colors of the edges don't matter in the distance graph, but I keep them around to help visually associate them with the STN constraints they're derived from.

The advantange of putting the STN in distance graph form, rather than encoding it as a set of linear constraints, is that we can leverage efficient shortest-paths algorithms to infer implied constraints. The notion of adding weights along a path's edges corresponds directly to "adding" the linear inequalities those edges represent. All the internal vertices, or event variables, "telescope" away, and we're left with an upper bound on the "temporal distance"

from the start of the path to the end.

For example, the path $L \xrightarrow{-0.5} K \xrightarrow{-14} B \xrightarrow{0} A$ tells us that A must happen *at most* -14.5 minutes after L. This sounds a little convoluted due to the negative duration, but it's easily rectified by reinterpreting it using positive time: The entire plan must finish at L *at least* 14.5 minutes after it starts at A. (It may help to imagine fixing A or L in time when reinterpreting.) While this could have been evident from the STN's visual depiction in Figure 4-13, the distance graph form establishes a uniform convention for the edge weights, making it easier to apply graph algorithms.

It turns out that an STN has a solution (i.e., is consistent) if and only if its distance graph has no negative cycles. And fortunately, we have the standard Bellman-Ford algorithm for single-source shortest paths (SSSP), which detects negative cycles. There exist many extensions to Bellman-Ford that improve its runtime for certain graph topologies [10], but they all have the same worst-case runtime of $O(nm)$, where n is the number of vertices, and m is the number of edges. Regardless, the point is that any one of those variants will detect and return a negative cycle if one exists. It's also important to note that *only one* such cycle will be returned, even if there are other negative cycles, and it will be a *simple* cycle. (Theoretically, returning *all* negative cycles is a much more expensive operation.)

In our example, there are two simple negative cycles: $B \xrightarrow{12} K \xrightarrow{-14} B$ and $D \xrightarrow{9} H \xrightarrow{-11.5} D$. Figure 4-15 shows only the former being detected and returned by an SSSP black box. This is the output of the subproblem, and now it is the master problem's job to translate this cycle into the risk allocation space.

Having received the negative cycle, the master observes that $12 + (-14) = -2 < 0$. But what it really needs is the cycle's weight in terms of the risk allocation variables that contributed to it having negative weight. The strategy is thus to trace back our derivation steps, going from the distance graph, back to the STN, back to the STNU, and finally back to the risk allocation that was applied to the pSTN.

Along this chain, the most consequential step is undoing the strong controllability reductions when reconstructing the STNU. Figure 4-16 reminds us that the edges in the negative cycle were derived from the STNU's $B \dashrightarrow C$ uncontrollable duration and $C \rightarrow K$ requirement. Namely, the weight on $B \xrightarrow{12} K$ is derived from the assignment $l_1 = 10$ plus

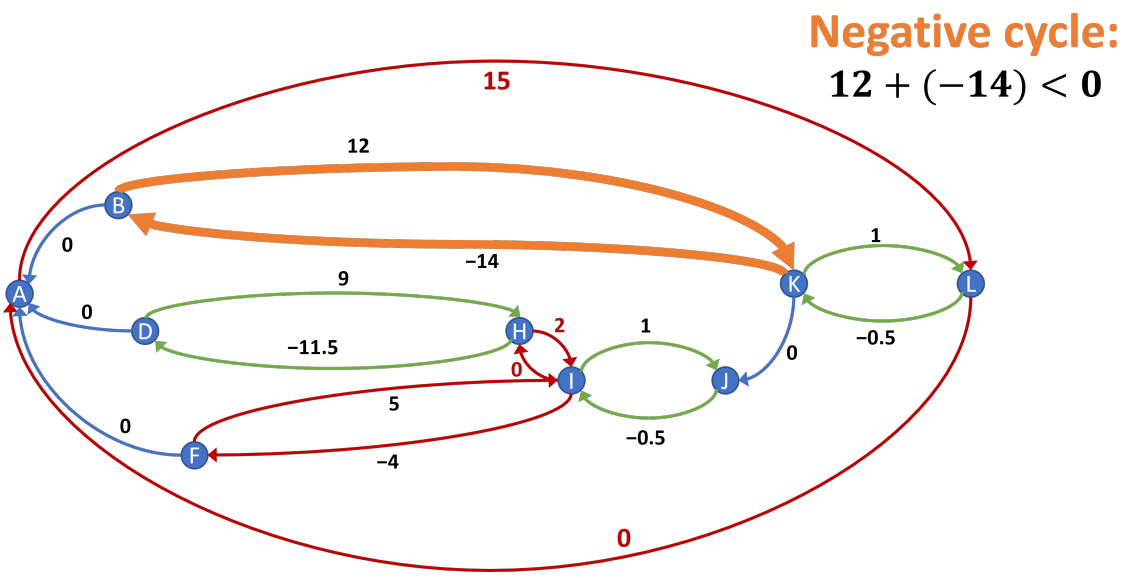

Figure 4-15: A negative cycles exists within the distance graph, composed of the two edges between B and K. Other negative cycles exist, too, but this is the one we focus on in the example.

the upper bound of 2 on $C \to K$. Similarly, the weight on $K \xrightarrow{-14} B$ is the *negative* (due to distance graph semantics) of the assignment $u_1 = 14$ plus the lower bound of 0 on $C \to K$.

Therefore, we can decompose the negative cycle's weight expression into $(10 + 2) + (-14 - 0) < 0$. All that's left to do is to "unassign" the risk allocation variables l_1 and u_1, and we get the following inequality for the cycle's weight:

$$\begin{aligned} (l_1 + 2) + (-u_1 - 0) &< 0 \\ \Leftrightarrow l_1 - u_1 + 2 &< 0. \end{aligned} \tag{4.7}$$

Equation 4.7 is what generalizes the grounded negative cycle in the distacne graph into a condition in the risk allocation space. Namely, it expresses precisely the condition needed for a risk allocation to make the distance graph cycle $B \to K \to B$ have negative weight.

Therefore, by negating that condition and enforcing it for all future risk allocations, we prevent that cycle from ever having negative weight again. *This* is the conflict resolution constraint that the master problem must learn.

$$l_1 - u_1 + 2 \geq 0. \tag{4.8}$$

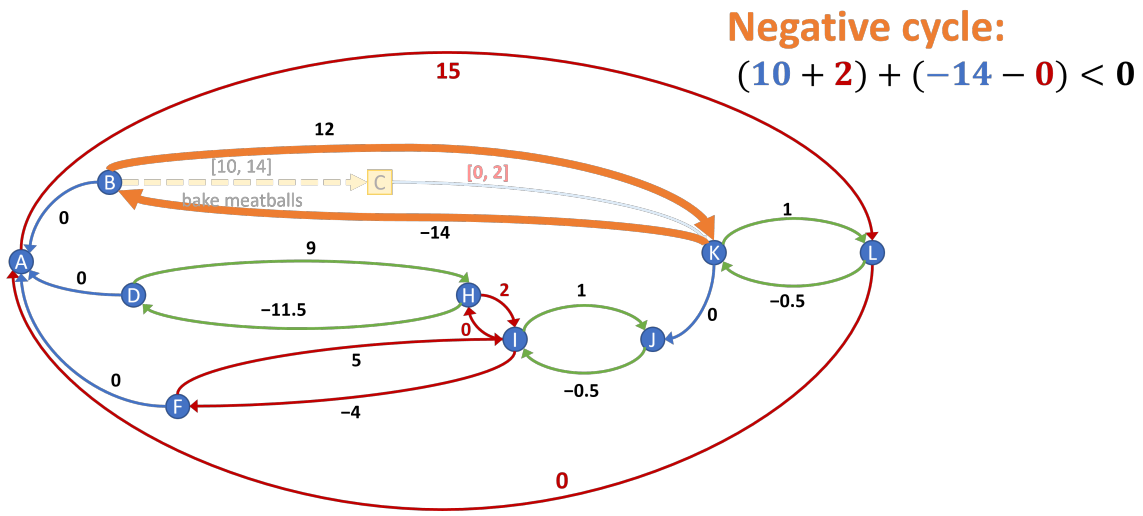

Figure 4-16: Recall that the weights of $B \xrightarrow{12} K$ and $K \xrightarrow{-14} B$ were derived from reducing requirement $C \rightarrow K$ through the uncontrollable duration $B \dashrightarrow C$.

Figure 4-17 depicts the conflict and its resolution in terms of the STNU that was provided as input to the subproblem, rather than the internally derived distance graph. The idea is that I've put the $B \dashrightarrow C$ and $C \rightarrow K$ *of the STNU* in distance graph form. I will elaborate on this concept of STNU distance graph in the next subsection on dynamic controllability conflicts. The gist for now, though, is that while $C \rightarrow K$ gets turned into an upper-bound forward edge and negative lower-bound backward edge, we do the opposite for the uncontrollable duration $B \dashrightarrow C$: We put its *lower bound* l_1 as the weight for the forward edge, and use the negative *upper bound* u_1 for the backward edge's weight. In a nutshell, the reasoning for the forward edge is that among all possible outcomes, l_1 represents the tightest upper bound on the difference in execution times $C - B$, and to be safe, a static schedule should respect that *a priori*. Similar reasoning can be applied for the backward edge weight of $-u_1$.

Therefore, we could alternatively argue that the negative cycle conflict is $B \xrightarrow{10} C \xrightarrow{2} K \xrightarrow{0} C \xrightarrow{-14} B$ in the STNU's distance graph. The disadvantage of doing so is that now the cycle isn't simple, which is a property that the SSSP algorithms rely on. We'd also have to create exceptions for negative cycles that shouldn't be considered conflicts. For instance $B \xrightarrow{10} C \xrightarrow{-14} B$ shouldn't be thought of as a conflict, because they come from the same uncontrollable duration. This cycle being negative is just an artifact of the distance graph

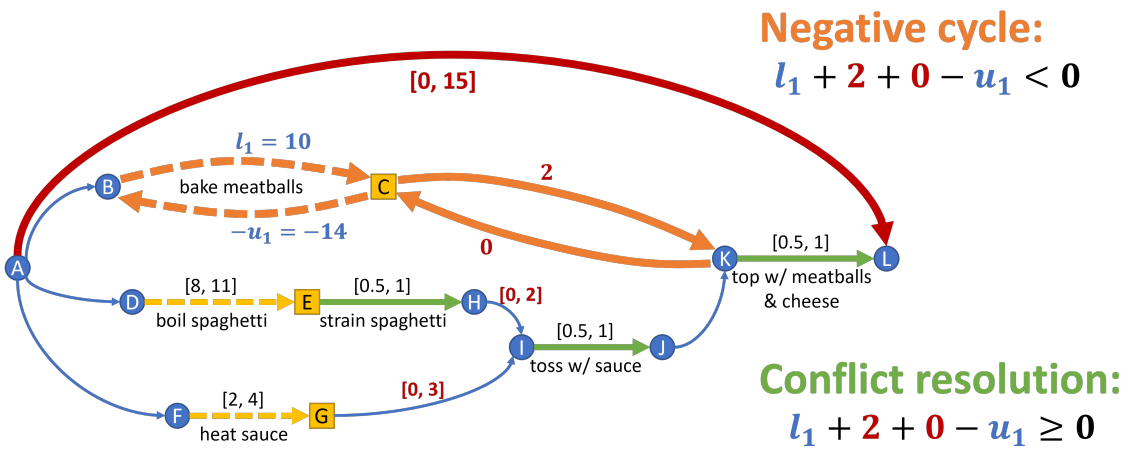

Figure 4-17: Tracing the edge weights in the negative cycle back to the STNU, we get a negative cycle in the STNU's own distance graph. "Unassigning" the risk allocation variables yields a cycle weight expression in terms of those variables. Our conflict resolution constraint for the master becomes enforcing that expression to be non-negative.

construction. Again, the following subsection will introduce mechanisms to handle that.

Ultimately, the semantics of strong controllability are rooted in the reduction to an STN. So it makes sense to operate on the STN when testing the subproblem's feasibility. It just means we need to maintain records of the various mappings along the way, so we can reconstruct and "lift" the negative cycle conflict into the risk allocation space.

For the remainder of this subsection, I will visually summarize how these conflict resolutions manifest in the risk allocation space, and relate them to the notion of Benders cuts from linear programming. Visually, the key observation is that Equation 4.8 is a linear inequality. That means we can envision it as a hyperplane cutting through the risk allocation space, depicted in Figure 4-18. Risk allocations on one side of the hyperplane correspond to making a particular cycle in the distance graph have negative weight. Therefore, we want to be on the other side, and treat the negative side as an obstacle.

Recall from Figure 3-10 that there is also a green region with nonlinear boundaries corresponding to the reformulated chance constraint. All candidate risk allocations generated by the master problem must land inside this green region. If the first candidate results in a not strongly controllable STNU, we will discover that it lies within a conflict region that is bounded by the hyperplane representing some cycle's weight being 0. In fact, the orthogonal distance of that candidate to the hyperplane is exactly how negative that cycle's

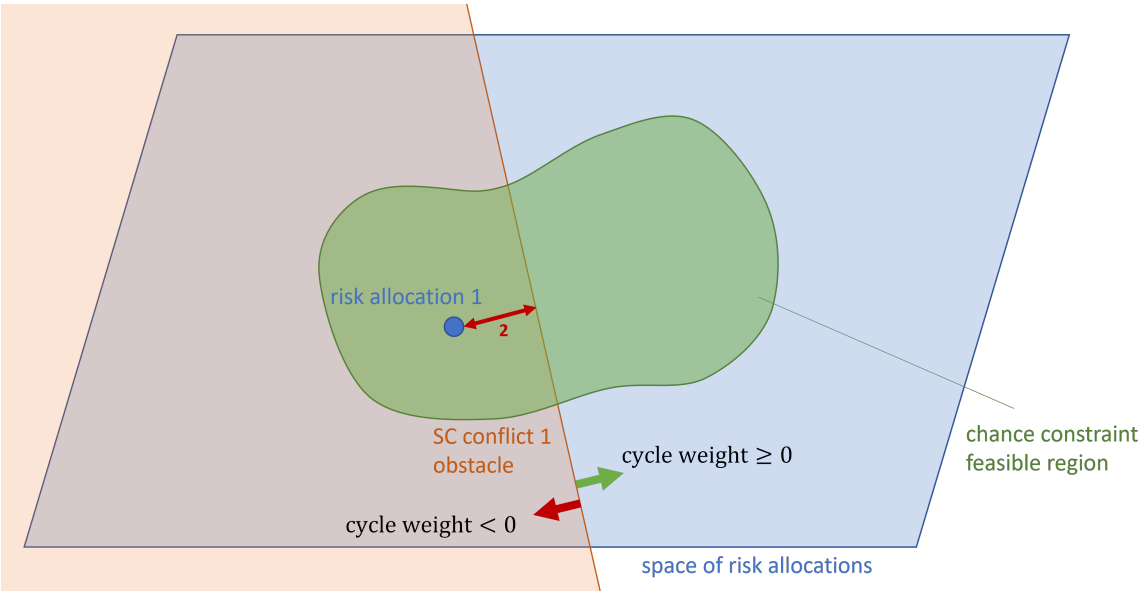

Figure 4-18: Because the cycle weight expression is linear in terms of the risk allocation variables, when the cycle's weight is 0, it forms a "line" (or hyperplane) in the risk allocation space. We want to be on the side of that line where the expression is non-negative, so the conflict resolution blocks out a half-plane (or half-volume).

weight is.

Since this first risk allocation lies within the obstacle, the obstacle must prune away some portion of the reformulated chance constraint region from further consideration. If there is still some green region left, then the master solver will pick another candidate from within it. For example, suppose a second risk allocation was generated where $(l_1, u_1) = (14, 15)$. In the STN distance graph, the previously negative cycle would become $B \xrightarrow{16} K \xrightarrow{-15} B$, so it no longer has negative weight. However, it would introduce *another* negative weight cycle $A \xrightarrow{15} L \xrightarrow{-0.5} K \xrightarrow{-15} B \xrightarrow{0} A$. If the subproblem returned this conflict, the master would learn its resolution as another hyperplane-bounded region, with the boundary 0.5 away from the second candidate. This is shown in Figure 4-19.

This process would continue until either the master generates a candidate that results in a controllable STNU, or the learned conflict resolutions entirely block out the green region, in which case the master would return infeasible. There is the question of whether this process is guaranteed to terminate. In other words, would we ever be in a situation where the conflict resolutions prune ever-shrinking infinitesimal pieces out of the green region for the master problem to select further risk allocations from. In Section 4.4, I prove this

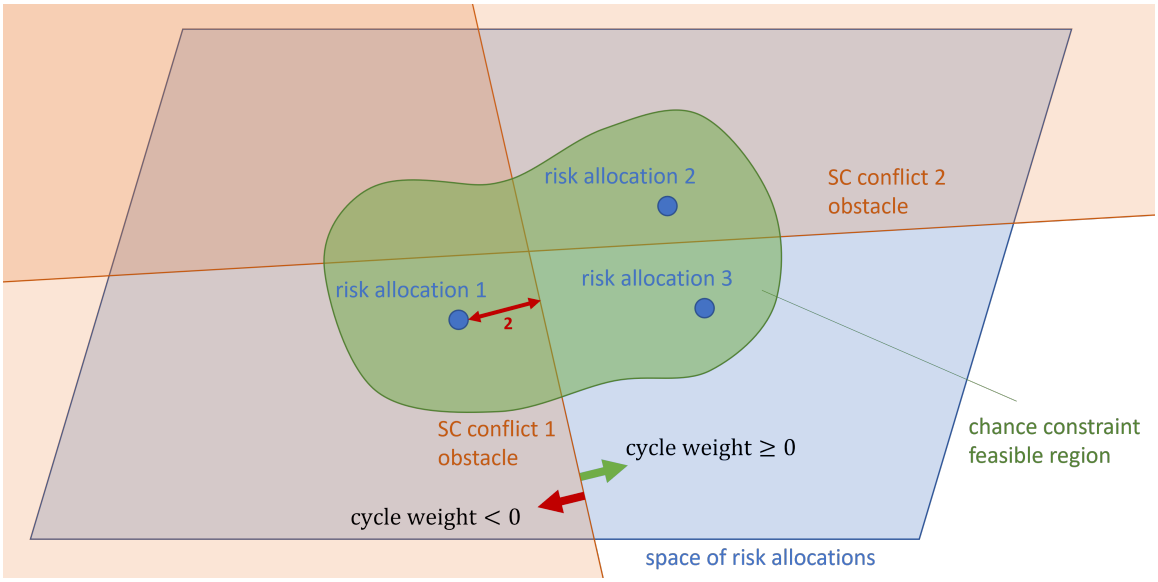

Figure 4-19: Subsequent risk allocations by the master must fall in the green chance constraint-satisfying region, which is ever-shrinking due to linear cuts by the negative cycle resolutions.

cannot be the case.

The final topic in this subsection is to relate these conflicts to the notion of *cuts* from Benders decomposition. Although I presented conflict-directed search upfront in the context of discrete variables, and then generalized its principles to our continuous-variable problem, Benders [4] had invented decades ago a similar technique in the context of mathematical programming. In his work, he allowed for mixed continuous and discrete variables, but he was only concerned with formulating subproblems that are linear programs over the continuous variables. He also allowed for an objective function in the overall master problem, as is the custom in mathematical programming.

Benders's key insight regarding the subproblem LPs was to leverage the concept of duality. Rather than solve the LP outright, he formulated and solved the LP's dual problem. Suppose that original problem's objective is to minimize a function over all the variables. Then the subproblem LP's objective would be to minimize that same function *with the non-LP variables assigned by the master*. The subproblem's dual LP would then have to *maximize* a related function, but also with the non-LP variables assigned. Once we solve the dual LP, then by strong duality, whatever maximum objective it arrived at would be a *lower bound* for the original problem's objective. Furthermore, by "unassigning" the non-

LP variables, we could generalize that lower bound as a function *across the entire non-LP variable space*. This is what's known as a *Benders cut*, and it becomes a new constraint for the master problem.

In our approach, the conflict resolutions learned by our master problem can be viewed as a special case of Benders cuts. First of all, our cc-pSTN problem is one of satisfiability, not optimization. This is easily accounted for in the Benders setting by making the objective function 0. Second, we are concerned with the case when the subproblem is infeasible. By duality, this means the subproblem dual would have a solution with an unbounded $+\infty$ objective value.

To interpret the Benders cut we get out of this, we need to understand what the form of the subproblem and its dual are. We already know, from Subsection 4.1.1, that our STNU strong controllability subproblem can be encoded as an LP, where each constraint has been compiled into a difference constraint relating two events. It turns out, as shown by Bhargava [5], that the dual of this LP expresses the notion that there can be no negative cycles in the compiled STN's distance graph. Namely, the objective function of the dual would be the *negative weight* of any path in the STN. If the STN has a negative cycle, then that objective gets pushed to $+\infty$.

The Benders cut extracted from this situation would express that the negative of the cycle's weight should be a lower bound of the original problem's objective function. But recall that objective is 0, so equivalently, we can say that 0 *should be a lower bound* on the (non-negated) cycle's weight. And thus, we have shown that the Benders decomposition interpretation of our problem results in exactly the same conflict resolutions learned by the master problem. Fang [19] makes note of this relationship in passing when citing prior work [69] that established this conflict-directed approach for producing static chance-constrained pSTN policies.

4.3.2 Dynamic controllability conflicts

The previous discussion of strong controllability conflicts and their resolutions in risk allocation space sets up much scaffolding that we can reuse when switching to dynamic

controllability. Namely, conflicts are still in the form of negative cycles, so the same principles apply when lifting cycle-weight expressions into the risk allocation space. The main differences are that the cycles will be in the distance graph of the STNU itself, and they will come with extra conditions attached. These cycles are known as what's called *semi-reducible*.

Thus, while we don't have to go through the step of compiling into an STN first, these semi-reducible conditions will introduce logical conjunctions into the mathematical statement of each DC conflict. By De Morgan's law, that means when we negate the expressions to form the conflict resolution constraints for the master, we'll get logical *disjunctions*. Each disjunct will be a linear inequality like the form of the strong controllability conflict resolutions. Therefore, resolving DC conflicts requires an additional layer of reasoning over that which is used to resolve SC conflicts.

As I said earlier in this section, this chapter is concerned only with establishing the form of DC conflicts and their resolutions. Chapter 5 will introduce the machinery needed to handle those resolutions. The remainder of this subsection will follow an arc similar to that of the previous: I will walk through an example of finding a semi-reducible negative cycle (SRNC). Then I write down the expression proving it's an SRNC, and it will be in conjunctive linear form. It thus remains to lift that expression into the risk allocation space, and negate it into a disjunctive linear form. Finally, I present the visual interpretation of these conflict resolutions, and I remark on potential relationships to extensions of classical Benders decomposition.

The key property used to find DC conflicts is that an STNU is dynamically controllable if and only if it has no SRNCs in its distance graph. This is known as the Fundamental Theorem of STNUs, originally discovered by Morris [38] and so named by Hunsberge [33], and it can be considered the counterpart to Fundamental Theorem of STNs. Unlike STN negative cycles, though, SRNCs are not necessarily simple.

The running example I will use is presented in Figure 4-20. This is the same spaghetti scenario as in Figures 4-10 and 4-11, but the risk allocation assignment being applied to the pSTN has $l_2 = 6$ instead of $l_2 = 8$. I claim that the portion of the network boxed in red forms an SRNC in its STNU distance graph form. Figure 4-21 isolates that portion of the

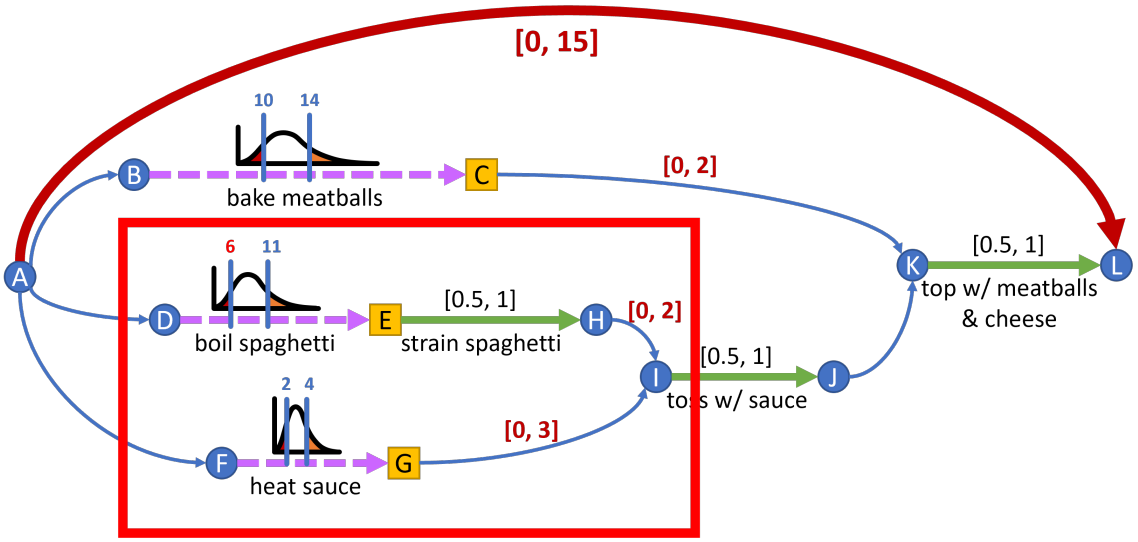

Figure 4-20: A risk allocation is generated for our pSTN running example. (Note that l_2 on $D \dashrightarrow E$ has been assigned 6 as opposed to previous instances where it was 8.) This subsection will focus on the highlighted subset of the pSTN.

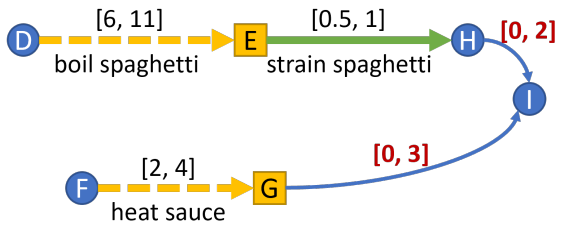

Figure 4-21: This is the same subset of the pSTN from Figure 4-20, now in grounded STNU form due to an assigned risk allocation.

STNU (the purple probabilistic durations have been replaced by yellow interval-bounded uncontrollable durations). Figure 4-22 then presents it in distance graph form.

The notions of an STNU's distance graph and semi-reducibility were presented in Subsection 4.1.2 when discussing the DC encoding. As a quick refresher, an STNU's distance graph is constructed according to the same principles as an STN's distance graph, with the exception of uncontrollable durations. Given such a duration $A \dashrightarrow B$ with bounds $[l, u]$, the forward edge in the distance graph becomes $A \xrightarrow{b:l} B$, and the backward edge is $B \xrightarrow{B:-u} A$. The lowercase b label means that l is the tightest possible constraint on the forward edge, but *conditioned* on the outcome of the $A \dashrightarrow B$ duration taking on its shortest value l. Likewise, the uppercase B label is a condition on $-u$ being the tightest

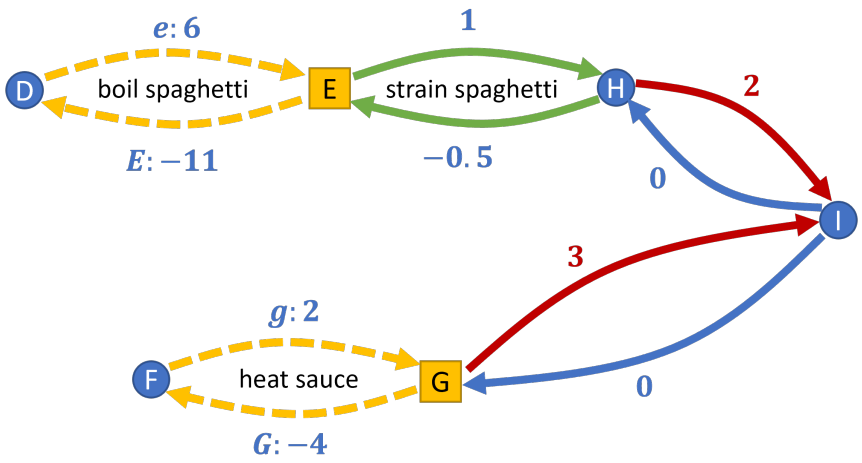

Figure 4-22: This is the distance graph form of the STNU segment in Figure 4-21.

possible constraint on the backward edge. The key idea is that these labels become moot during execution once the respective outcomes actually appear in the execution history. But before then, and thus during offline analysis of controllability, it is essential to respect those constraints.

It turns out there is an asymmetry in how we handle lowercase versus uppercase edges. Because lowercase outcomes represent the *earliest* possible time an uncontrollable duration $A \dashrightarrow B$ could finish, a scheduling policy has to *unconditionally* assume that outcome. If it didn't make that assumption and it turns out the outcome was indeed l, then if some event C needed to execute before B arrives, the policy might have accidentally dispatched C too late. In contrast, uppercase outcomes are not subject to such scrutiny, because in these situations, if the actual outcome was earlier, that actually *relaxes* the deadlines for other events' execution. Therefore, the policy can actually react to those outcomes, by virtue of being able to affect the future but not the past.

This asymmetry is what the DC reduction rules in Subsection 4.1.2 are trying to express: The lowercase reduction rules result in new edges without the lowercase label, whereas the rules involving uppercase labels preserve the label on the new edges (with the exception of the uppercase-removal rule). In essence, the lowercase edges are compiled away just like in strong controllability, but the uppercase edges are conditionally propagated. This then, is the notion of *semi-reducibility* for a path, that there exists a sequence of reductions than can be applied to adjacent edges, ultimately resulting in a path with no lowercase edges.

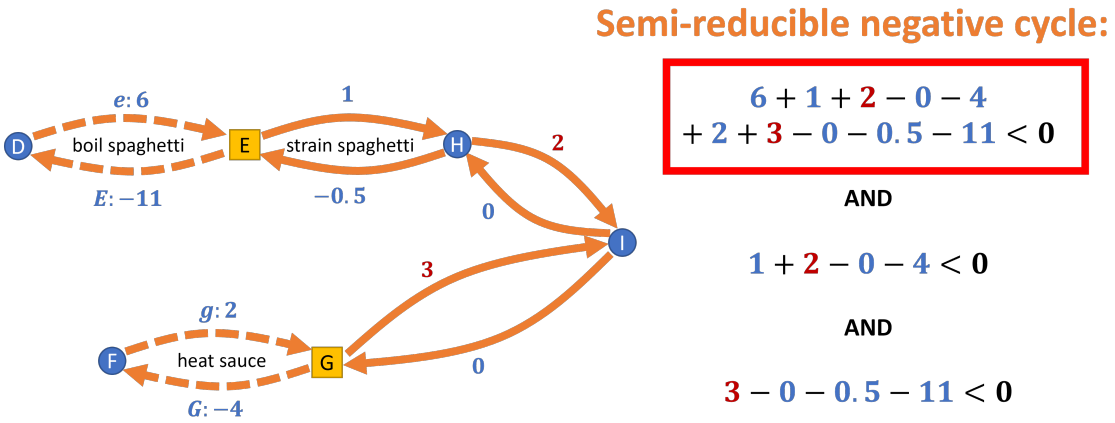

Figure 4-23: The entire segment in Figure 4-22 forms a semi-reducible negative cycle. Here we note that the cycle's weight is indeed negative.

What this amount to is that the compiled distance graph necessary to begin dynamically dispatching an STNU contains all the ordinary and uppercase-labeled edges, but drops the original lowercase edges. This final form of the distance graph, which Morris calls the *AllMax projection* [41], must not contain any negative cycles. If there *is* a negative cycle, then we can trace in reverse the reduction rules that yielded any derived edges. We will finally obtain a cycle in the original, non-compiled distance graph, and that cycle is an SRNC, offering proof of the STNU being not dynamically uncontrollable.

Figures 4-23 through 4-25 demonstrate that the distance graph segment in Figure 4-22 is indeed an SRNC. First, we verify that the entire cycle has a negative weight of -1.5. Then, we need to make sure that $D \xrightarrow{e:6} E$ and $F \xrightarrow{g:2} G$ can be compiled away by application of the lowercase or cross-case reduction rule.

Let's begin with $D \xrightarrow{e:6} E$. The claim that Figure 4-24 makes is that a series of reductions apply to the path $E \xrightarrow{1} H \xrightarrow{2} I \xrightarrow{0} G \xrightarrow{G:-4} F$, transforming it into a new edge $E \rightarrow F$ with negative weight. This path is called an *extension subpath* for $D \xrightarrow{e:6} E$, and is what allows the lowercase compilation to occur [38]. Recall from Subsection 4.1.2 that the DC reduction rules all take on a plus-minus form. This extension subpath has the property that the last edge weight is more negative than all the previous edge weights combined. Therefore, we can apply reduction rules from the end of the extension, yielding a sequence of new edges $I \xrightarrow{G:-4} F$, $H \xrightarrow{G:-2} F$, and $E \xrightarrow{G:-1} F$. Since the magnitude of this last edge

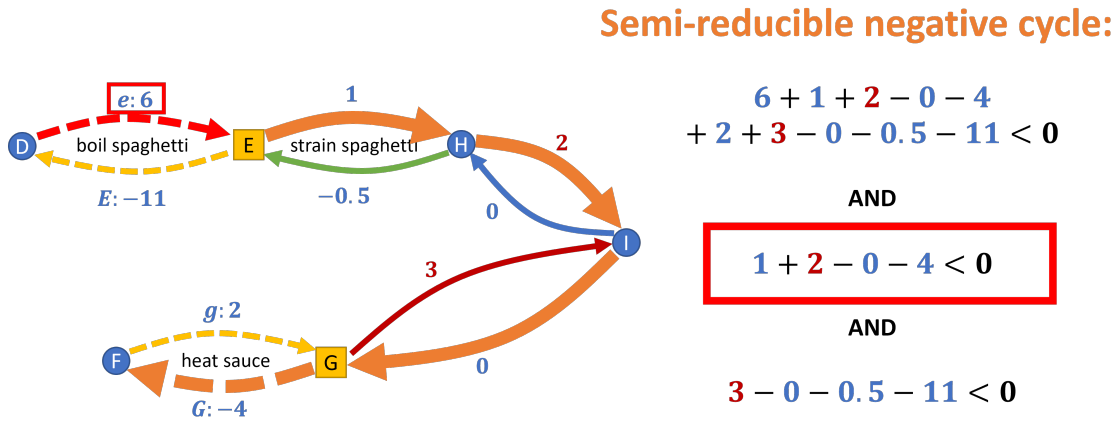

Figure 4-24: The first condition attached to the SRNC is the fact that lowercase edge $D \xrightarrow{e:6} E$ can be reduced against the negative-weight path $E \xrightarrow{1} H \xrightarrow{2} I \xrightarrow{0} G \xrightarrow{G:-4} F$. The reduction would yield an ordinary edge $D \xrightarrow{5} F$.

weight is less than the shortest duration of 2 for the $F \to G$ activity, the label removal rule applies, and we're left with $E \xrightarrow{-1} F$. Finally then, we can reduce that against $D \xrightarrow{e:6} E$ to yield $D \xrightarrow{5} F$.

The key condition that allowed this final reduction to happen was that the extension subpath had a negative total weight. We also had to check that there existed a series of reductions on the extension subpath itself to get it down to one edge. It turns out that as long as the extension contains no other lowercase edges, and it ends on a negative weight edge, then it can be reduced to a single edge.[3] The only negative-weight edges are either the negative upper bounds of uncontrollable durations, or simply negative-weight ordinary edges. Since those edges will always be unconditionally negative for any candidate STNU, there's no value in encoding that. Therefore, the main condition for semi-reducibility is that the extension paths have negative weight, and this is identified in Figure 4-24.

To complete the example, we consider the extension subpath for $F \xrightarrow{g:2} G$, shown in Figure 4-25. In this case, we can't reduce from the end because both $E \xrightarrow{E:-11} D$ and $H \xrightarrow{-0.5} E$ are negative. Instead we reduce from E backwards, yielding $I \xrightarrow{-0.5} E$ and then $G \xrightarrow{2.5} E$. Now this is positive, so we reduce against $E \xrightarrow{E:-11} D$ to get $G \xrightarrow{E:-8.5} D$.

[3]Technically, the extension could contain *nested* lowercase edges and their own extensions. However, that would mean they could be reduced/compiled away into new ordinary or uppercase edges, which would then be used in the reductions for the containing extension subpath.

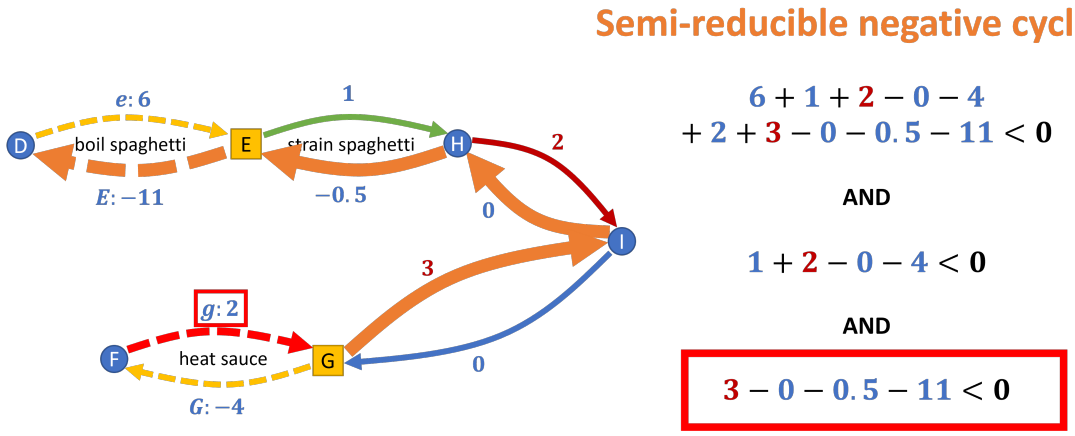

Figure 4-25: Likewise, lowercase edge $F \xrightarrow{g:2} G$ can be reduced against the negative-weight path $G \xrightarrow{3} I \xrightarrow{0} H \xrightarrow{-0.5} E \xrightarrow{E:-11} D$. This yields an uppercase edge $F \xrightarrow{E:-6.5} D$.

Finally, the cross-case rule applies to $F \xrightarrow{g:2} G \xrightarrow{E:-8.5} D$, and we get $F \xrightarrow{E:-6.5} D$.

In summary, the SRNC in Figures 4-23 through 4-25 reduces down to $D \xrightarrow{5} F \xrightarrow{E:-6.5} D$, which is composed entirely of ordinary and uppercase edges. Thus, we have identified a *conjunction* of conditions that verify a) the cycle has negative weight, and b) the lowercase edges can be compiled away.

The remaining steps are to lift this expression into the risk allocation space, and then negate it to form a conflict resolution constraint for the master. Fortunately, we already know how to perform most of this from our experience with strong controllability conflicts. We simply identify any terms in the grounded conflict that came from assignments to risk allocation variables, and then "unassign" them, putting in their place the variables themselves. Thus, we have a conjunction over multiple inequality expressions. This is shown in Figure 4-26. The first conjunct is a linear inequality that says that the cycle weight is negative. All subsequent conjuncts are then saying that *certain subpaths of the cycle* have negative weight themselves, so they are linear inequality as well.

Now, instead of negating a single linear inequality, we are negating a conjunction of them. Since we'd prefer not to deal with the "not" of an "or" explicitly, we push the negation onto the linear inequality conjuncts themselves, which we know how to do. De Morgan's law thus requires us to replace the conjunction with a disjunction, so the conflict resolution is expressed as a disjunction of linear inequalities with the inequality sign flipped. This is

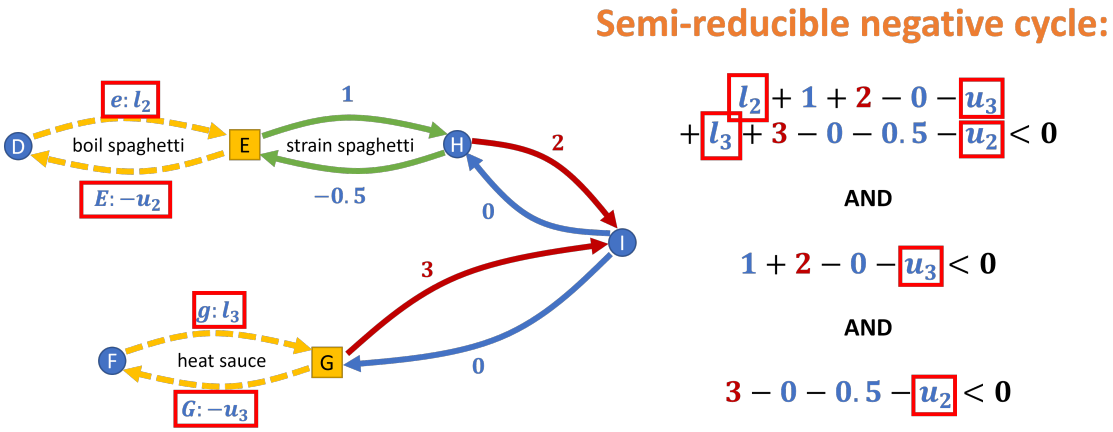

Figure 4-26: To lift the SRNC conflict expression into the risk allocation space, we simply replace the terms that correspond to risk allocation variables with the variables themselves. This gives us a conjunction of linear inequalities.

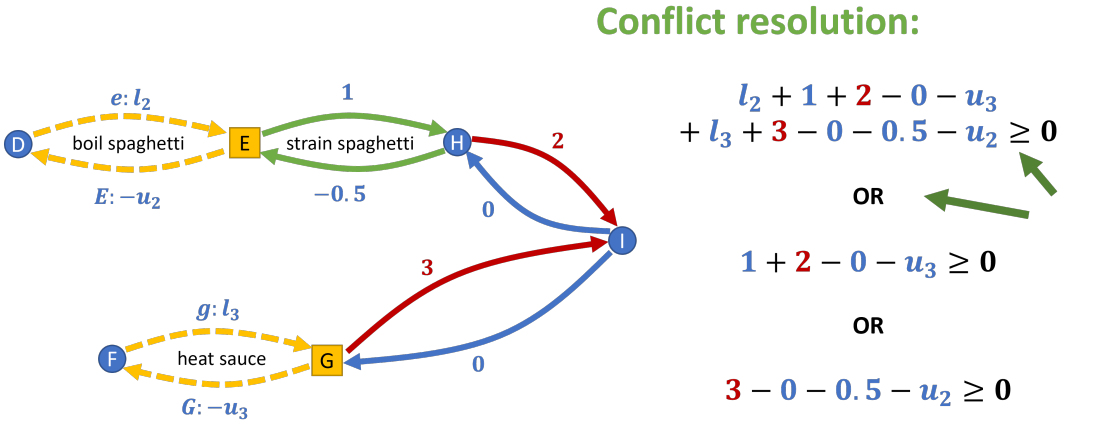

Figure 4-27: To resolve the conflict, we apply De Morgan's law to flip the inequality signs *and* turn the conjunction into a disjunction.

shown in Figure 4-27. In essence, what this says is to avoid the SRNC conflict in future risk allocations, we can either make the cycle's weight negative, *or* we can make non-negative the weight of any of the extension subpaths, thus preventing the cycle from becoming semi-reducible.

There are two edge cases to note about SRNC conflict resolutions: First, an extension subpath becomes a useful disjunct only if it contains risk allocation variables in its weight expression. If it doesn't, then that extension can never change its weight, so there's no value in telling the master problem to try to raise it. Second, if a negative cycle in the STNU's (non-compiled) distance graph contains no lowercase edges, then there are no extension

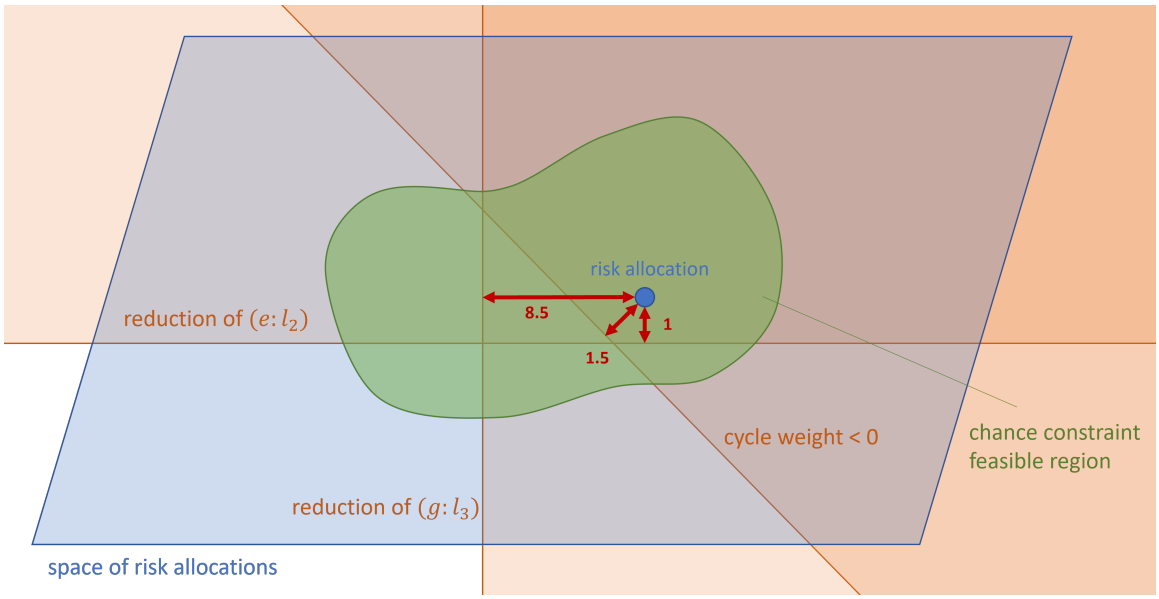

Figure 4-28: Each disjunct in a disjunctive linear conflict resolution cuts through the risk allocation space on a hyperplane, with one side being feasible and the other not. The risk allocation that led to the SRNC must fall within the infeasible side of each hyperplane. Its distances to the hyperplane boundaries are exactly how negative the weights of the cycle and the lowercase-edge path extensions were.

subpaths. In this case, the cycle remains in the AllMax graph, and it can be treated like the negative cycles for strong controllability conflicts. That is, we can simply flip its inequality sign, and not bother encapsulating it in a logical operator. In general though, we assume DC conflict require disjunctive resolutions.

What do these disjunctive linear conflict resolutions mean for the risk allocation space? Figure 4-28 provides insight. Since each disjunct is a linear inequality, we can interpret it the same way as a strong controllability conflict resolution. That is, it specifies a hyperplane on which one side the inequality is violated. In Figure 4-27, we had three distinct inequalities, each with a different combination of risk allocation variables. Threfore, we get three cuts through the risk allocation space, all with different "slope".

The distance of the risk allocation to the hyperplane boundaries is exactly the amounts by which the weights of the cycle and the extension paths were negative. Moreover, since the conditions of the SRNC violated *all* of the disjuncts, the risk allocation must fall within the violating side of all the hyperplanes. Thus, this identifies a region that is the intersection of all the half-volumes bounded by the hyperplanes. This region is precisely the new obstacle

Figure 4-29: To satisfy the disjunctive linear constraint, one can view its negation as the obstacle in risk allocation space. The negation is exactly the intersection of all the hyperplane-bounded obstacles. Hence, it is a *convex polytope* obstacle.

in the risk allocation space that the master problem must avoid from now on, shown in Figure 4-29.

Like when we were searching for static policies, this obstacle prunes away a portion of the remaining green region representing the reformulated chance constraint. However, the options to remove any one of the SRNC's extension subpaths make it a *convex polytope* obstacle. This means it has the potential to prune less of the chance constraint region, leaving more possibilities for the next candidate. For example, if we had only considered making the cycle's weight non-negative, and ignored the extension paths, then the obstacle in Figure 4-29 would have been a half-volume bounded by the diagonal hyperplane.

Now that we know the form of DC conflict resolution obstacles, we can generalize the diagram in Figure 4-19 to Figure 4-30. For each risk allocation that didn't pass the STNU-checking subsolver, instead of learning a half-volume obstacle, the master problem learns a convex polytope. Although the diagram is only illustrative and not "to-scale" with respect to any particular SRNCs, the key insight we get is that each DC obstacle might block out less of the chance constraint region than SC obstacles do. This is consistent with our intuitive hypothesis that we expect more cc-pSTNs to admit dynamic policy solutions

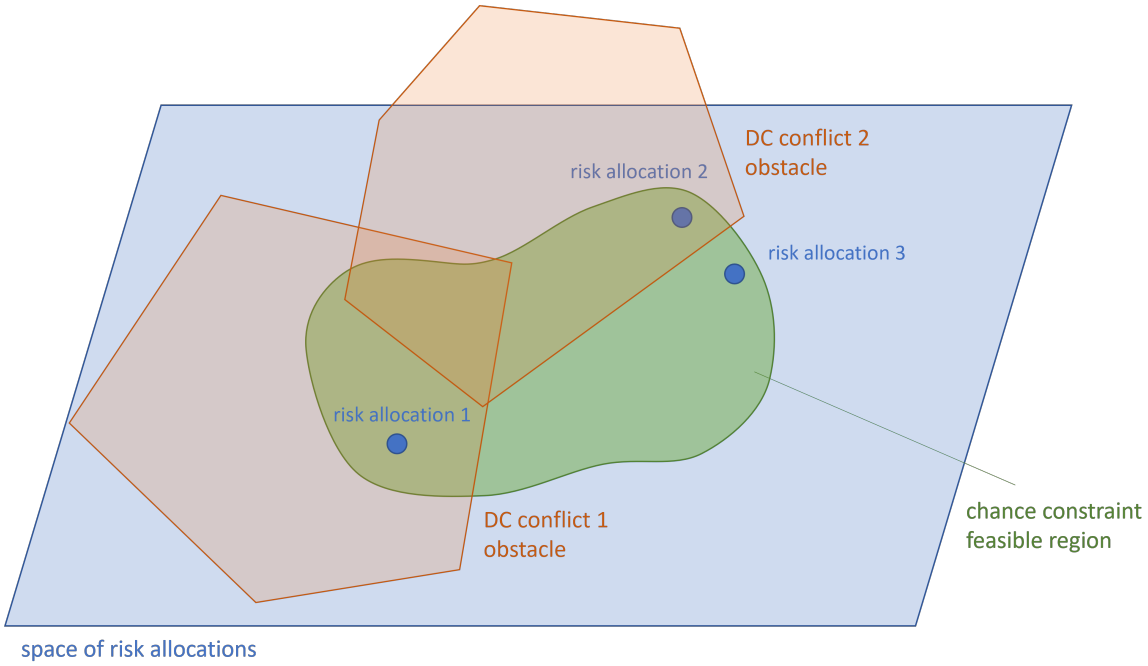

Figure 4-30: A conflict-directed solver for chance-constrained dynamic policies would discover a convex polytope obstacle for each candidate STNU that's not dynamically controllable. These obstacles are tighter than the half-volume obstacles in Figure 4-19. This explains why dynamic policies have a potentially larger solution space than static policies.

than only static policies, just as dynamically controllable STNUs are a superset of strongly controllable ones.

When we were searching for static policies, I claimed that the conflict-discovery process must eventually terminate, because there are a finite number of topologies for strong controllability conflicts. The same argument applies for dynamic controllability conflicts, but it is harder to prove, because as we saw, SRNCs are not necessarily simple cycles. Fortunately, it turns out that the way SRNCs are discovered by dynamic controllability-checking algorithms implies they also have a finite number of topologies. There are simply many more of them than those for SC conflicts. This will be fully argued in Section 4.4.

For dynamic policies, there remains the question of whether we can find a solver for the master problem that can actually process these disjunctive linear constraints along with the reformulated chance constraint. Unfortunately, the presence of convex *obstacles* means that the remaining feasible space is *non-convex*, and so this prevents us from leverage efficient convex solvers. This is unlike the case for SC conflict resolutions, where the complement of

a hyperplane-bounded half-volume takes the same form, and therefore preserves convexity of the solution space.

Ultimately, as I noted at the beginning of this Section 4.3, the master problem for generating dynamic policies faces the same difficulty as the full DC encoding does in the form of constraints it has to solve. It's just that the conflict-directed approach typically requires fewer such disjunctive linear constraints, and each of them is more informative in encompassing entire SRNCs rather than individual edges. Chapter 5 will present the final pieces needed to solve our master problem.

Like with the last subsection, I close by discussing the relation of our DC conflicts to other work and Benders cuts. The concept of resolving STNU dynamic controllability conflicts was first introduced by Yu [74]. This was done in the context of *relaxing* overconstrained temporal plans. Whereas prior work in plan relaxation would remove activities or requirements entirely, Yu's insight was to keep them but adjust their bounds, thus retaining the cycles but making them either non-negative or not semi-reducible. Therefore, he faced the same issue of disjunctive choices for resolving a DC conflict.

The main difference is that work assumed an STNU model, not a pSTN, so there was no nonlinear chance constraint. However, it did include an objective function indicating preference or cost for degrees of relaxation. For the purposes of experimentation, linear cost functions were used, thus reducing the features required of the underlying solver. Additionally, I expand on their work by giving a deeper argument about the structure of DC conflicts and their resolutions, plus the geometrical interpretation.

In theory, we should be able to carry over our interpretation of SC conflict resolutions as Benders cuts to DC conflict resolutions. The complicating factor, though, is that our subproblem can no longer be encoded as a linear program. Recall that Benders cuts are derived from the objective value of the dual problem's solution. Whereas duality is well-understood for LPs, the question is whether we can formulate a dual problem for the disjunctive linear constraint program given in Subsection 4.1.2. If we could, then we would expect the expression for an semi-reducible cycle to fall out of the dual's objective function, just as the objective for an STN's dual LP gives us a cycle weight expression.

Currently, no one has yet tried to formulate this dual problem for the DC encoding. To be

sure, it would be significantly more complex than the SC encoding's dual. However, there are some promising extensions to the classical Benders decomposition that may provide the key insights forward. The most useful insight comes from Hooker [29], who summarized the concept of an *inference dual* program and how Benders decomposition leverages the dual's solution in the abstract. He then applied it to the classical propositional satisfiability (SAT) problem, partitioning the Boolean variables into two sets. The result, which he called *logic-based Benders decomposion*, creates Benders cuts that are disjunctions of literals.

Imagine, then, if in our disjunctive linear DC encoding, we introduced Boolean variables to indicate the truth- or falsehood of each linear disjunct. The encoding could then be framed as a SAT modulo theories (SMT) problem, where the theory is that of linear difference constraints. This suggests that the theory would return conflicts that string individual difference constraints into negative-weight cycles or paths, and then the framework similar to logic-based Benders decomposition could them combine them disjunctively into an SRNC expression. To be sure, this is a conjecture and likely contains subtle errors, but if something similar is true, it would highlight a deep connection that frames DC conflict resolutions as Benders cuts.

An alternate way to reach the same conclusion might come from Geoffrion's work, which generalizes classical Benders decomposition to settings where the subproblem could be nonlinear program [23]. In that work, the dual program from which Benders cuts are extracted is formulated via nonlinear convex duality theory. I mentioned on page 109 that there exists a nonlinear programming encoding of the DC conditions, albeit less efficient. Nevertheless, from a theoretical standpoint, it may also be possible to extract SRNC conflict expressions by applying Geoffrion's work to that NLP encoding.

4.4 Incompleteness revisited

Throughout this chapter, we have argued that the policies we output meet all the conditions of our reformulated Problem 3.8. Particularly, in Section 4.2, we constructed Algorithm 4.1 such that any solution is derived from a risk allocation that satisfies the chance constraint *and* yields a controllable STNU, which is exactly what Problem 3.8 requires. Hence,

Algorithm 4.1 is sound.

Lemma 4.1 (Soundness)**.** *Algorithm 4.1 is sound with respect to the reformulated Problem 3.8.*

It is less straightforward to demonstrate it is complete. We know, however, there are only two possible exit conditions: either the solver in line 7 couldn't find a risk allocation solution, or the STNU was found to be controllable in line 11. Furthermore, since discovered conflicts' resolutions are necessary conditions for STNU controllability, if we terminate due to them making the risk allocation constraint program infeasible, then we know that no such risk allocation solution exists for the overall Problem 3.8.

In other words, the algorithm is sound with respect to problem instances that have no solution. Intuitively, the figures in Subsections 4.3.1 and 4.3.2 demonstrate this by how the conflict resolutions block out portions of the risk allocation space. When they entirely block out the green chance constraint region, we know that it is safe to return infeasible.

Therefore, as long as we show that Algorithm 4.1 always terminates, it follows that it terminates with the correct answer, and so it is sound and complete with respect to the reformulated problem. We address termination separately for static and dynamic policies. The overall argument for both is the same, though. As previously indicated on page 129, since each iteration discovers a new negative cycle, it suffices to show that the number of such cycle topologies is bounded.

For static policies, we receive STNU strong controllability conflicts, which in turn are derived from the compiled STN form. Namely, every such conflict is detected as a negative cycle in the STN's distance graph. The Bellman-Ford algorithm is commonly used for this purpose, and it has the property that *if* a negative cycle exists, then it will discover *some* negative cycle that is *simple*, i.e. the cycle contains no repetitions of vertices. (The nuance here is that Bellman-Ford doesn't necessarily discover the cycle we had in mind, which might not be simple, but there exists at least one simple negative cycle, and Bellman-Ford will detect one of them.) Clearly, there are a finite number of simple cycles in any finite distance graph, even if it is exponential in the number of vertices. Therefore, we will eventually run out of negative cycles to discover.

For dynamic policies, the argument is slightly more involved, because STNU dynamic controllability conflicts, which are semi-reducible negative cycles (SRNCs) aren't necessarily simple cycles. Morris provides an example of this [38]. However, he continues with a crucial observation that if an STNU contains an SRNC, then there exists (possibly another) SRNC of limited complexity, the form of which we explain shortly. Hence, when checking DC, we can limit ourselves to semi-reducible cycles of that form. All DC-checking algorithms effectively rely on this observation, which means there are a finite number of such cycles they could discover. This is thus the parallel argument to the static case, whereby we replace the restriction of looking only for simple cycles with that of looking for semi-reducible cycles with a bounded structure.

This structure is argued as follows: If a semi-reducible path (and hence any cycle) contains a lowercase edge, then by definition, that lowercase edge is followed by a subpath, called an *extension*, which allows the lowercase edge to be reduced away. Recursively, though, that extension must be semi-reducible as well, so we need to consider if it contains any other lowercase edges, as well as their extensions nested within. This could be problematic if the extension for some lowercase edge, say $A \xrightarrow{b:3} B$, contains that same edge inside of it, because this could lead to infinite nesting.

Morris's observation is that the SRNCs we need to look for *don't* have such nested repetitions. Therefore, since there are K lowercase edges total, there can be at most $K-1$ other lowercase edges within a top-level extension subpath in a semi-reducible path. By induction on the nesting, starting from the innermost extension, we can conclude that the top-level extension has a bounded number of edges. Hence, the SRNCs we consider must consist of some number of top-level lowercase edges and their finite extensions, plus any additional ordinary and uppercase edges interspersed in between.

The final step in arguing these SRNCs are bounded in size is to show that each lowercase edge only needs to appear at most once as a top-level edge. Since the cycle is semi-reducible, we will be able to reduce top-level lowercase edge against its extension, resulting in a cycle consisting of only ordinary and uppercase edges. We can then apply the same reasoning regarding simple cycles in the STN unlabeled distance graphs, and conclude that this cycle, if not simple, must also have a simple nested cycle. We then undo all the reductions to

restore the top-level lowercase edges, and we reach the contradiction that if such an edge shows up twice, then the reduced cycle we restored from cannot be simple.

Therefore, for both strong and dynamic controllability, the negative cycle conflicts we receive from the checker are restricted in their structure and bounded in the number of edges they could contain, with respect to the size of the STNU. This means our conflict-directed algorithm cannot continue discovering such conflicts forever, and must eventually terminate, as Lemma 4.2 below states. Together, Lemmas 4.1 and 4.2 imply that our conflict-directed algorithm solves the reformulated problem exactly.

Lemma 4.2 (Termination)**.** *Algorithm 4.1 is guaranteed to terminate.*

Theorem 4.3 (Correctness)**.** *Algorithm 4.1 is sound and complete with respect to the reformulated Problem 3.8.*

In practice, the completeness of the algorithm depends on that of the NLP and STNU subsolvers, which so far we have implicitly assumed are fully correct. The literature on STNU controllability checking has produced several algorithms with rigorous proofs that they are sound and complete [33] [39]. Completeness is harder to guarantee for NLP, because algorithms for solving them typically step through a continuous space and may get stuck in local features. For instance, the Ipopt solver we use in our implementation uses an interior-point method, and when it can't find a feasible region in the variable space, it returns a message saying the problem seems to be locally infeasible [64].

The user of our algorithm has a couple options to reduce the chances of encountering such incompleteness from the NLP solver. First, if we stick with unimodal PDF distributions when modeling the probabilistic durations, it's likely that the reformulated chance constraint will be convex, and we can thus employ convex programming solvers, which generally have stronger guarantees of convergence, and not to mention better runtimes. The justification for convexity can be seen in back in Figure 3-2 and Definition 3.1, when we use the CDF function to define the risk of cutting off tails. When the PDF is increasing, the CDF is convex, and vice versa. Therefore, as long as the chosen lower and upper bounds are on opposite sides of the mode, then the risk expression will be convex, and so will the total sum of individual durations' risks in the reformulated chance constrained, expressed in

Definition 3.5 and Equation 3.3.

Second, when calling the NLP solver, we can supply an initial point that is reasonably close to the actual feasible region. A simple choice would be the uniform risk allocation, where the risk bound is distributed evenly across all $2K$ tails of the K probabilistic durations. For more sophisication, we could use the previous risk allocation solution as the initial point for the next round. That solution now lies in the infeasible region, due to the recently learned conflict resolution, but the hope is that if it's not too far from the new boundary, then it won't take much effort to cross into the remaining feasible region.

In conclusion, our algorithm is still incomplete with respect to the original cc-pSTN problem. However, it introduces no new incompleteness on top of the reformulated problem, with the exception of the NLP solver's incompleteness, which can be mitigated to a reasonable degree.

Chapter 5

Hierarchical Algorithm Design

Chapter 4 presented a high-level algorithmic strategy, illustrated by Figure 4-9, for finding chance-constrained scheduling policies using conflict-directed search. This strategy relies on supporting routines to identify STNU controllability conflicts, and then resolves them in the risk allocation master problem. We were able to identify the form of these conflicts and their resolutions in both the strong and dynamic controllability cases, which respectively yield static and dynamic chance-constrained policies. Thus, we have a unified framework that encompasses previous work [69], which only produced static policies.

However, in the dynamic case, the conflict is a semi-reducible negative cycle (SRNC), whose validity is predicated on the existence of negative-weight extension subpaths for any lowercase edges. Thus, there may exist alternate ways to resolve such a conflict, by making the weight of such extensions nonnegative. The DC conflict resolutions express these alternatives using *disjunctive* linear constraints.

Any further risk allocation must therefore satisfy at least one of the disjuncts from each such conflict resolution, so our algorithm needs to be able to handle these disjunctions. Unfortunately, using a black-box solver to handle those *in combination with* our nonlinear reformulated chance constraint would be impractical. We would have to turn to mixed-integer nonlinear programming (MINLP), which would pull in much generalized machinery we may not need.

Instead, we take inspiration from disjunctive linear programming (DLP), and apply its core technique of branching on disjuncts to yield a sequence of linear programs (LPs).

For our problem of resolving DC conflicts within the risk allocation space, we replace these LP subproblems with that of risk allocation. These risk allocation subproblems necessarily include the nonlinear chance constraint, plus all the implied constraints of STNU controllability.

To focus on framing these risk allocation subproblems, we employ the simplest branching strategy, which is presented upfront in Section 5.1. This illustrates a key insight that lets us build our solver for dynamic policies as a layer *on top of* the existing solution to the static policy problem, rather than replacing the static version's master solver. The idea is to leverage the relationship between SC and DC conflicts presented in Chapter 4 so that by "activating" a single disjunct for each disjunctive conflict resolution, we reduce the dynamic version of the master problem back into a form that matches the static version's. Solving for chance-constrained dynamic policies, then, reduces to performing combinatorial search over the disjuncts.

The ultimate contribution of this chapter is to implement this strategy as a collection of interacting algorithms that solve a *layered hierarchy* of subproblems. Framing it this way decomposes and extends the conflict-directed strategy of Algorithm 4.1. Our claim is that the existing static policy solution can be framed as two layers interacting, and the dynamic policy solution as a third layer on top.

The value in this framing is that we can identify common principles that apply to each layer. That is, each layer is responsible for solving the constraints handed to it and framing subproblems for the layer below, while returning conflicts to the layer above. Therefore, rather than having a two separate monolithic algorithms in the style of Algorithm 4.1, we arrive at a decomposable architecture that illuminates not just the structural similarities in the discovered conflict, but also the computational similarities in how they are resolved.

Section 5.1 presents the intuition of our algorithmic strategy to resolve DC conflicts. Then, Section 5.2 identifies the three subproblems that comprise what the three layers of our dynamic policy-generating algorithm are solving. Section 5.3 gives us an algorithmic template for addressing these layers in the abstract. Finally, Sections 5.4 and 5.5 instantiates this template for each layer. The end result is a collection of functions that compose to form the full algorithmic solution.

5.1 Branching on DC conflict disjuncts

By themselves, solving the nonlinear chance constraint and solving the set of disjunctive linear conflict resolutions do not pose significant computational challenges. The former can be addressed by NLP solvers [25] [64]. The latter can be solved by DLP solvers [34] [36], or even framed as a mixed-logic linear program (MLLP) and sent to an appropriate solver [28]. However, it's the combination of solving both types of constraints at once that is not well-studied.

Since our disjunctive constraints form a DLP, it is worth considering the techniques that DLP solvers employ to handle this structure. The foundational method is to perform search by branching on the disjuncts. That is, we select one disjunct to be "active" from each disjunction. The key observation is that when branching, we "tighten"[1] a disjunctive constraint into just a single linear constraint. This means once we've branched on all disjunctions, we're left with a linear program to solve.

Thus, we can imagine a search tree that performs the branching, and each leaf represents an LP. If we exhaustively traverse this tree, and solve the LP at each branch, this is a sound and complete strategy for solving the original DLP. If one of the leaves yields a solution, then it must be a solution to the DLP, because each disjunction is satisfied by the linear constraint that was branched on. Conversely, if the DLP has a solution, then we can identify at least one disjunct per disjunction that it satisfies. Stringing together the disjuncts, we can thus find a leaf whose LP is satisfied by the DLP's solution.

This is excellent news for us, because if we ourselves construct such a search tree on our learned disjunctive conflict resolutions, and append our nonlinear chance constraint to the LPs at the leaves, then we have a form that the static version of our conflict-directed approach can solve. Thus, our viable strategy is to replicate the combinatorial search framework of DLP, so that we can attach our custom nonlinear chance constraint, and replace the LP solver at the leaves with an NLP solver. It is worth noting that this separation of higher-level

[1]The constraint programming literature often calls this "relaxation", even though it actually expands the obstacle representing the constraint, and thus reduces the remaining feasible space, which technically "tightens" the problem. This terminology is likely related to the use of "relaxation" in shortest-paths algorithms when applying the triangle rule [12]. The idea is that if the tighter constraint is satisfied, then there is no more "pressure" to satisfy the original one, so it can now "relax".

combinatorial search and lower-level reasoning of constraints over continuous variables is reminiscent of SMT solving, which was mentioned in the previous two chapters. This insight will be key to helping us frame a "third-layer" subproblem in Section 5.2, which specifies the role that our combinatorial search approach will be fulfulling.

To illustrate this approach, I visually elaborate on how the DC obstacles in Figure 4-30 would be processed. Recall that each linear inequality disjunct of a DC conflict resolution corresponds to a hyperplane blocking off a half-volume in the risk allocation space. Since the convex polytope obstacle is the intersection of all those half-volumes, the "facets" of the obstacles are effectively specified by those hyperplanes. Therefore, branching on a disjunct means choosing a facet, and "expanding" it back out into an infinite hyperplane.

This is shown in Figure 5-1, where we have generated a first risk allocation, but found the implied STNU was not dynamically controllable, and so we extracted a conflict. The master problem generalizes the conflict into a convex polytope obstacle containing the risk allocation. We proceed by choosing the "top" facet to branch on.

We note two aspects of this branching. First, the expansion turns the original obstacle into a linearly-bounded obstacle that entirely contains the original. Therefore, it potentially removes from consideration a larger chunk of the chance constraint-satisfying region. This accounts for the soundness argument: if a second risk allocation can be found in the remaining green region *and* it yields a dynamically controllable STNU, then that risk allocation respects the original convex polytope obstacle as well.

However, it also has the potential to overprune risk allocations which might have otherwise succeeded. So second, this is why we need branching for completeness. If we eventually branch on every facet of this obstacle, expanding them in turn, then all areas of the green region not covered by the obstacle will be considered at least once by the risk allocation generator.

At this stage in our example, we have chosen to branch on the "top" facet of the first obstacle, and this leaves some green region remaining. Thus our NLP solver will be able to find another risk allocation. Suppose like the first one, it also turns out to yield an uncontrollable STNU. Then we learn another convex polytope obstacle, this time surrounding the second risk allocation, as shown in Figure 5-2.

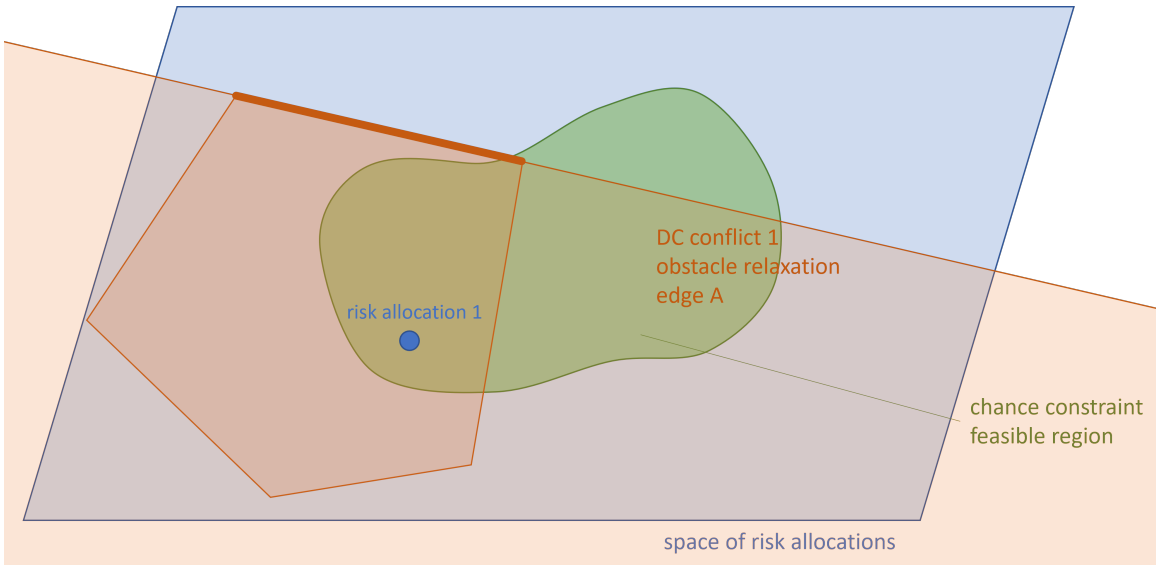

Figure 5-1: When the first DC conflict is found, we branch on the edges, or "facets", of its convex polytope. Each facet represents a linear inequality constraint. In this diagram, we are branching on the top facet, so this "tightens" the DC constraint, or equivalently "expands" the obstacle into a half-volume of the risk allocation space.

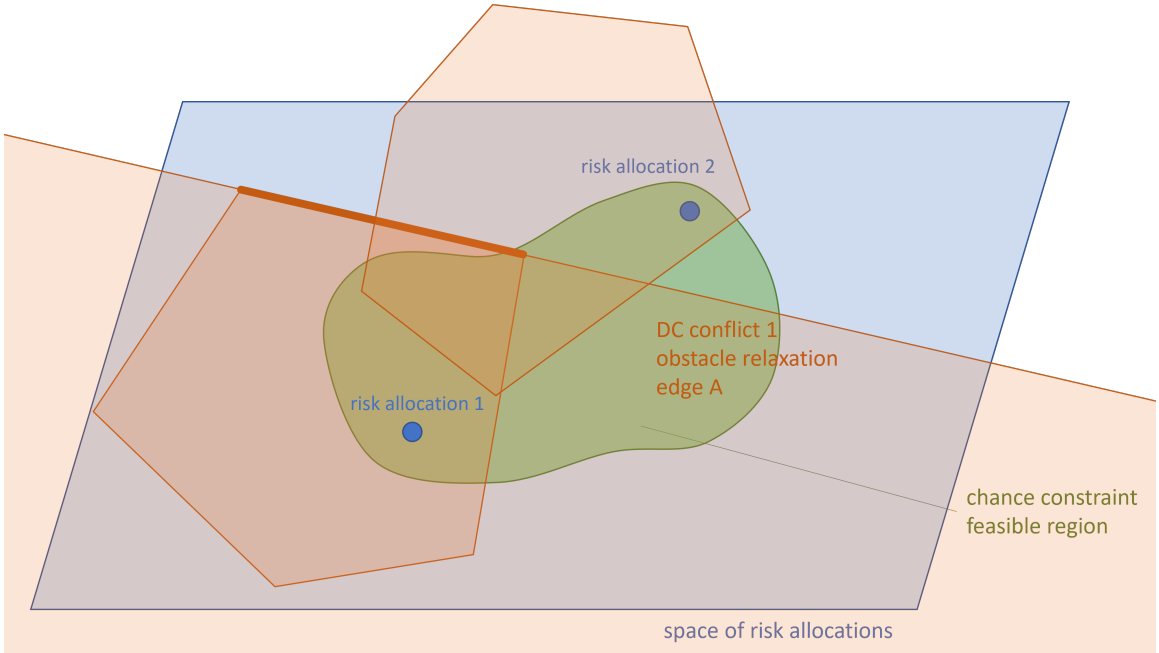

Figure 5-2: In the remaining space of the chance constraint region, the master chooses a second candidate risk allocation. If this yields an STNU that is not dynamically controllable, then another convex polytope obstacle is discovered.

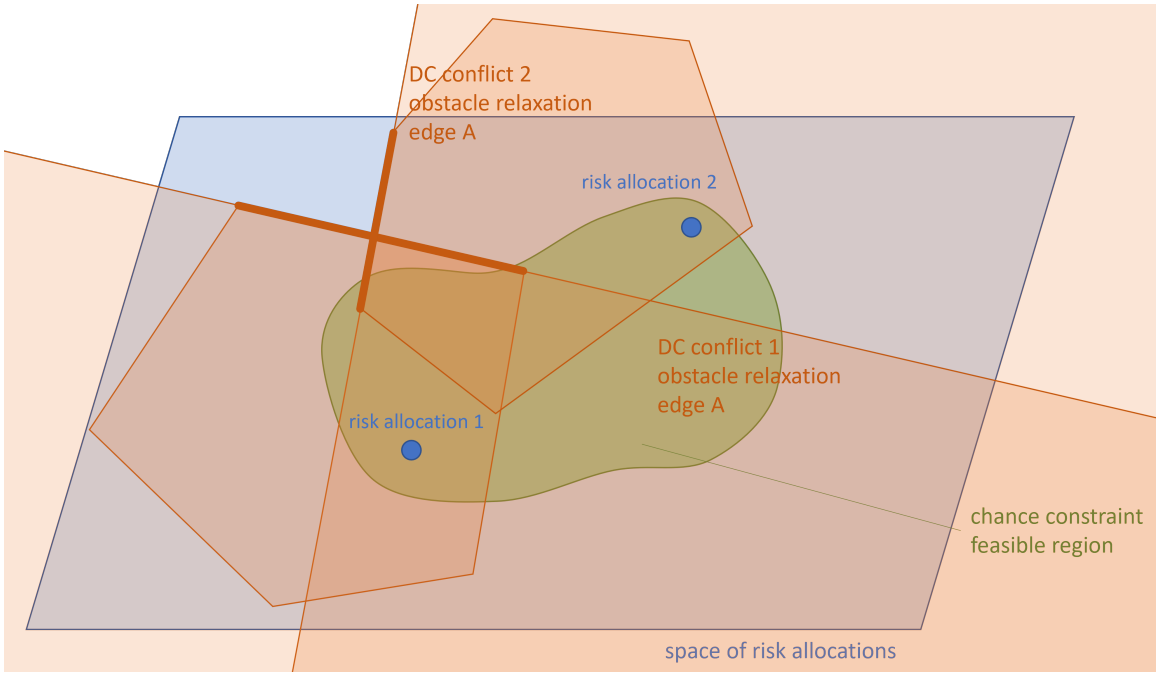

Figure 5-3: Suppose for obstacle 2, we pick the leftmost facet to branch on first. Then combined with obstacle 1's facet that we had branched on earlier, they entirely block out the chance constraint region. Therefore, no risk allocation can be generated, and we must backtrack and try other combinations of facets to branch on.

Now we have the responsibility of branching on this second obstacle's facets. Suppose we pick the "leftmost" facet first. Note that this branching is performed *under* the first branch of the first obstacle. Thus we collectively block out the two half-volumes shown in Figure 5-3. Unfortunately, in this case, the chance constraint feasible region is entirely contained within these two half-volumes. What this means is that the branch is a dead end, and so we have to backtrack.

This is the key difference between our algorithmic solutions for generating dynamic versus static chance-constrained policies. In the static case, if we had blocked out the entire chance constraint region, then we would immediately terminate and say there is no feasible policy. And that would be correct because our SC conflicts are unconditional negative cycles. But since SRNCs are conditioned on the lowercase edges being reducible by their extension subpaths, we can resolve them not only by enforcing those cycles to have non-negative weight, but also by making at least one lowercase edge non-reducible. These choices thus correspond to the different facets of the SRNC's convex polytope obstacle.

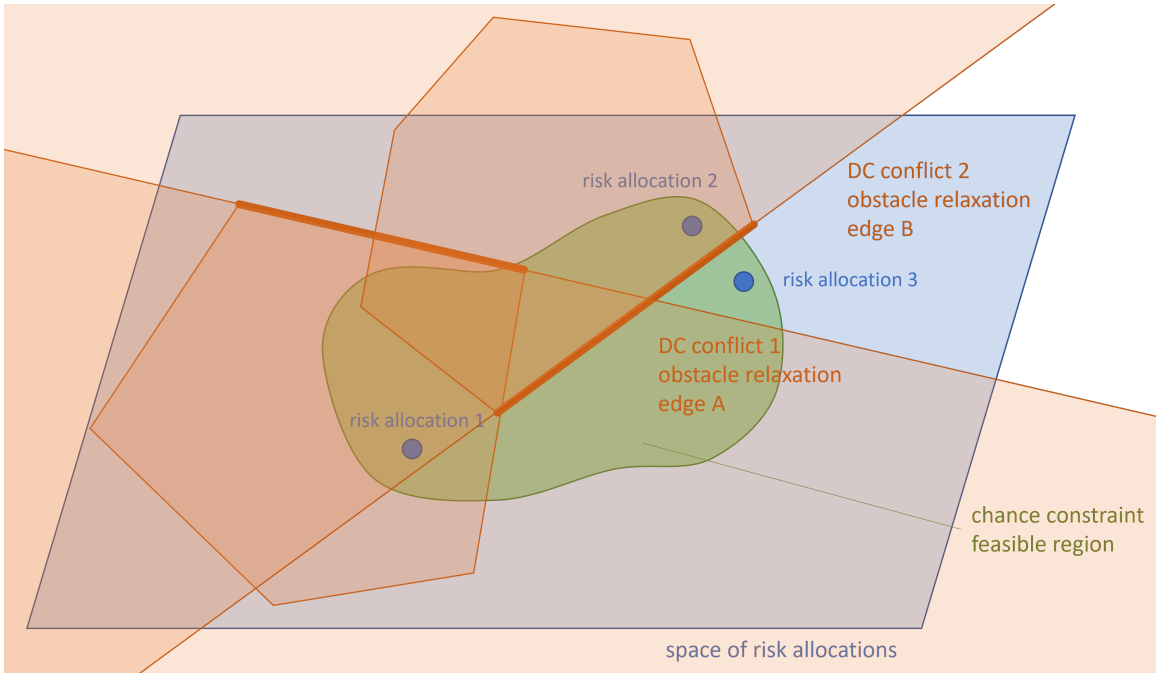

Figure 5-4: If we try branching on the bottom right facet of obstacle 2, then that leaves a small slice of the green region, from which the master can choose a third candidate risk allocation. Thus, we can continue checking for STNU dynamic controllability, and we will find either a valid solution, or an SRNC conflict.

Hence, for the dynamic case, when the NLP solver returns infeasible because there is no remaining chance constraint region, we still have to try other combinations of facets from each obstacle. Only when *all* combinations have been tried or pruned, and shown infeasible, do we conclude that we cannot find a dynamic chance-constrained policy.

To wrap up this example, let's say we backtrack on the first facet we branched on for the second obstacle, and then try branching on the "bottom-right" facet, shown in Figure 5-4.[2] Once we expand that next facet, and combine it with the expanded facet from the first obstacle, we see that there is a small sliver of the chance constraint region remaining. Thus, the NLP solver will be able to find a risk allocation, and we proceed by checking the dynamic controllability of its implied STNU, just as we did for the first and second risk allocations.

The takeaway here is that every time we successfully find a risk allocation, it must yield either a feasible policy or a conflict that becomes a new obstacle in the risk allocation space. This is the same as what happens in the static case, but because we require a layer of

[2]For the completeness argument, it's not crucial that we follow any particular branching order, as long as all facets are explored eventually. In this example, I chose this particular next facet for illustrative purposes.

branching on top of our existing architecture for finding static policies, these obstacles have implications for that layer. Namely, every DC conflict obstacle introduces a new choice[3] that has to be made before we've reached a leaf in our search tree that resolves all discovered conflicts. Therefore, the depth of our search tree grows with each DC conflict we find.

Since we're borrowing from DLP solvers this notion of performing high-level combinatorial search, I conclude this section by noting that this is only the most basic principle. DLP solvers typically employ additional strategies to speed up the search over their logical components, *prior* to running an LP solver at the leaves [3] [36] [28]. For instance, logical inference procedures in the categories of resolution or propagation can cut down on the number of choices that need to be made. This, in turn, can reduce the number of leaves by a multiplicative or even exponential factor.

For our solution approach, logical inference could manifest in the following ways: If we branch on a facet from a convex polytope obstacle, and the resulting blocked-off half-volume entirely contains another convex polytope obstacle, then there is no need to branch on that other obstacle's facets; it's already "relaxed" by the application of the current facet. This shaves off an entire level within the current subtree of our search. To be sure, determining whether any remaining obstacles fall within a half-volume incurs computational cost. One would have to identify all of their "extremal points", which could be a large number for high-dimensional obstacles.

A simpler, though less complete, method would be to simply check if the currently selected facet is shared with any other obstacles. This could happen if the facet represented the extension subpath of a particular lowercase edge, and that edge plus extension was also present in another discovered SRNC conflict. This is, in fact, the general strategy of Yu [74] when branching on DC conflict resolutions.

Another form of inference comes from the idea of conflict extraction *at the level of the nonlinear solver* and not just STNU controllability. Whenever we find that the disjuncts we've branched on completely block out our the chance constraint feasible region, we could try to determine what subset of them were responsible for that infeasibility. That is, we'd

[3]Recall that some SRNC conflicts may not contain any lowercase edges. In these cases, their resolution constraints are unconditionally linear, and so no branching is required. This distinction will be addressed in Section 5.5.

be looking for an irreducible infeasible subset of the chosen linear disjuncts plus the chance constraint. The chance constraint is an unconditional aspect of our problem, so we can't get rid of it, but we could make sure that future branches on the search tree don't replicate this combination of disjuncts.

To express this as a conflict, we simply gather those disjuncts into a conjunction, and so the resolution constraint is a disjunction over the *negations* of the offending disjuncts, which are still linear constraints That means our existing branching strategy for handling SRNC conflicts can also accept these conflicts. This actually increases the depth of branching required in the search tree, but when combined with the propagation procedures, branching on one of those disjuncts will automatically prune the associated disjuncts from the other convex polytope obstacles, thus reducing the branching factor across multiple levels.

Li [36] explores similar ideas of conflict extraction in the general DLP setting, which includes an objective function, but no nonlinear chance constraint. Her conflicts thus come from an LP solver, and increasingly bound the objective value. This is reminiscent of classical Benders cuts. Fang [19] generalizes the MLLP, which is itself a generalization of DLP, by incorporating a chance constraint, turning it into a cc-MLLP problem. His approach, however, is to avoid the computational expense of NLP solvers, and rather solve a series of relaxed LP problems. Under the reasonable assumption that the chance constraint region is convex, his strategy generates hyperplanes that provide outer-approximations of the chance constraint. When a candidate is chosen, but turns out to be infeasible or non-optimal, tighter hyperplanes are found.

Because the focus of this chapter is to create a layered architecture of subproblems that can all be solved by common algorithmic principles, I have chosen not to incorporate most of the above strategies for speeding up the combinatorial search. That is, in the spirit of avoiding premature optimization, the discrete search strategy I instantiate in the following sections is limited to simply branching on each facet of each discovered SRNC obstacle, as shown in the previous figures. Nevertheless, since the architecture I present separates out key patterns and roles, this makes it clearer where such additional strategies could fit in. Section 6.4 discusses related work along these lines.

5.2 Subproblem hierarchy

The previous section illustrated that we can resolve dynamic controllability conflicts, and hence produce dynamic policies, by branching on the polytope facets of the conflict resolutions. In doing so, we temporarily reduce, or "relax", the problem to one that contains only linear constraints plus the nonlinear chance constraint. This means that rather than substitute a completely new solver in line 7 of Algorithm 4.1 – which using an NLP solver satisfactorily produces static policies – we can largely preserve Algorithm 4.1 in that form and use it as a subroutine within a parent algorithm that performs the branching.

To define that parent algorithm, we need to specify what problem it is solving, namely, what its inputs and outputs are. We will also have to tweak the current inputs and outputs of Algorithm 4.1 so that the parent algorithm can receive disjunctive SRNC conflict resolutions, and send back down the linear disjuncts that were branched on. The key insight is that this relationship between the desired parent and the existing algorithm closely mirrors the master-and-subproblem relationship we designed for the latter!

That is, the parent algorithm can be viewed as solving the "master problem" for producing chance-constrained dynamic policies. Within the parent, it frames risk allocation subproblems for an algorithm similar to Algorithm 4.1 to solve. And as we know from Chapters 3 and 4 already, the risk allocation algorithm further separates out the subproblem of checking STNU controllability.

This section, then, is dedicated to specifying this three-layer hierarchy of master problem and subproblem relationships. If we have algorithms that fulfill each of these problems' input and output specification, then they will automatically combine to solve our reformulated cc-pSTN problem. This is what the remaining sections of this chapter provide. In preparation, Section 5.3 presents an algorithmic template for each layer to play its role as master problem to the layer below, and subproblem to the layer above. Sections 5.4 and 5.5 then instantiate this template at the requisite layers for static and dynamic policies, respectively. We are thus able to distill and unify the algorithmic principles linking the two variants of our problem.

In this rest of this section, I define the problem inputs and outputs for each layer, which I call a *level*. I proceed in a top-down fashion, starting with a common "top level", which is

simply responsible for performing the problem reformulation presented in Chapter 3. Then, I present Level 3, which is used only to generate dynamic policies by branching on SRNC conflict resolution disjuncts. Next is Level 2, which is the main machinery for supplying risk allocations. Finally, Level 1 is responsible for checking STNU controllability.

As the "user-facing" component of our architecture, the top level promises to solve the original cc-pSTN problem statement as given by Problem 2.20. Therefore, it simply accepts a pSTN, a chance constraint, and whether we are looking for a static or a dynamic policy. Problem 5.1 sets this up for us.

Problem 5.1 (Top-level problem reformulation)**.** *Given a cc-pSTN, find a scheduling policy that satisfies the chance constraint, or identify a subset of the cc-pSTN that prevents us from being able to do so.*

Inputs:

- *A pSTN* $\mathcal{N}^p = \langle \mathcal{E}, \mathcal{E}^u, \mathcal{A}, \mathcal{A}^p, \mathcal{R} \rangle$
- *A chance constraint* $c = \langle \mathcal{R}, \Delta \rangle$
- *Whether we want a static or dynamic policy*

Outputs:

- *Whether a scheduling policy satisfying the chance constraint was found*
- *If yes, an actual policy* $\mathcal{P}$
- *If yes, a risk allocation* $\mathcal{L} = \langle l_1, u_1, \ldots, l_K, u_K \rangle$ *as proof of the policy's correctness*
- *If no, a conflicting subset of the inputs*
- *If no, (optionally) an expression in terms of the conflicting subset, evaluating to false*

Ideally, if we succeed, then we are able to return an appropriate scheduling policy. However, since we know that the policies we find are ultimately derived from a proxy STNU, in practice all we need is the STNU. Online dispatching algorithms can then effectively generate any valid policy [39] [33]. For the purposes of conceptual exposition, I will continue to refer to

the output solution as a scheduling policy $\mathcal{P}$. As a certificate of the policy's correctness, we also supply the risk allocation that was found, since that uniquely determines the implied STNU from which the policy was derived.

If we have a negative result, then we would like to return a "cc-pSTN" conflict as an analogue to the SC and DC conflicts of STNUs. While this might not be as elegant as a single negative cycle, owing to the complicating nature of the chance constraint, we can abstractly say that a conflict is simply a subset of the input. Any other cc-pSTN with that same subset (e.g., certain activities or requirements plus the chance constraint) would be unschedulable for the same reason. Therefore, the top level's specification allows for a broad definition of cc-pSTN conflict, and we leave it to the implementing algorithm to decide what form that takes.

Like in the affirmative case, we allow for a certificate that shows why the conflict is a valid reason for infeasibility. Again, the abstract idea is that there is some expression *in terms of* the conflicting elements which cannot be satisfied. In the case of STN conflicts, for example, we need all cycles to have non-negative weight, which a conflicting cycle violates.

Internally, the top level reformulates the original problem (Problem 2.20) into that of risk allocation (Problem 3.8). Then it calls either Level 3 or Level 2 to solve it, depending on whether we requested a dynamic or static policy. This is demonstrated in Algorithm 5.1 at the end of this section. For now, we continue with the problem statements for the remaining levels.

Level 3, as we just said, is responsible for finding a risk allocation that yields a dynamic policy. Therefore, it takes in the original pSTN plus the reformulated chance constraint, which is constructed by the top level. Since this reformulation is expressed in terms of the risk allocation variables, those are passed in as well, so that a *full assignment* to them can be returned as the desired solution. Problem 5.2 specifies these inputs.

Problem 5.2 (Level 3 branching on disjunctive SRNC conflict resolutions)**.** *Given a pSTN and a reformulated chance constraint* rcc*, find an assignment to the risk allocation variables that satisfies the* rcc and *yields a dynamically controllable STNU. If unable to, return a risk allocation conflict.*

Inputs:

- *A pSTN* $\mathcal{N}^p$
- *A set of risk allocation variables* $\{l_1, u_1, \ldots, l_K, u_K\}$
- *A reformulated chance constraint* rcc *defined over those variables*

Outputs:

- *Whether a satisfying risk allocation was found*
- *If yes, an assignment to the risk allocation variables*
- *If yes, the STNU* $\mathcal{N}^u$ *implied by the risk allocation, and a feasible dynamic scheduling policy* $\mathcal{P}$ *for it*
- *If yes, (optionally) a set of linear conflict resolution constraints as proof of a feasible NLP when combined with the* rcc
- *If no, a set of learned SRNC conflict resolution constraints*

The output of Level 3 is to say whether a risk allocation satisfying the conditions of the reformulated problem was found. If so, that means the risk allocation implies an STNU that is dynamically controllable via some policy. For conceptual completeness, we return all three of these "objects" to the top level. A user's implementation of the top level can then decide which "form" of the solution to use; e.g., it could return the actual policy (represented as some sort of lambda function), or just the STNU, or even construct another policy according to a custom preference function.

Also, since the risk allocation assignment is ultimately generated by an NLP solver in Level 2, we can return that nonlinear program as a certificate of the risk allocation's validity. In fact, we don't need to include the reformulated chance constraint, since that's given. We can just return all the other constraints in the nonlinear program, which are just *linear* conflict resolution constraints. The exact sources of those constraints will be made clear in Section 5.5.

When Level 3 isn't able to find a risk allocation, that's because it has collected from Level 2 a set of disjunctive SRNC conflict resolution constraints, and determined that

their polytopes completely block out the chance constraint feasible region. Therefore, that collection of polytopes (represented algebraically) is in conflict with the reformulated chance constraint. We thus return that as Level 3's conflict to the top level, where it will be packaged into a cc-pSTN conflict.

I note that it could be possible there exists a *subset* of the polytopes which block out the chance constraint region. We know that that the last polytope discovered must be in that subset, because without it, a risk allocation was still found. But for example, if ten DC conflicts were found total, maybe the last polytope and only three of the first nine block out the region. Thus, these four polytopes would yield a smaller conflict, and thus be more informative to return. As I mentioned at the end of Section 4.2, though, performing this kind of analysis to make the conflict set minimal is out-of-scope for this thesis.

The last observation here is that although I motivated and claimed Level 3 is needed to perform combinatorial search over the polytopes' facets, the notion of search does not appear in the problem statement. This is because from an external point of view, i.e., the top level's, Level 3's job is to solve the reformulated problem for dynamic policies. Internally, Level 3 receives the polytopes to branch over as conflicts from Level 2. Thus, it constructs its own search problem, rather than being given it by the user. Sections 5.3 and 5.5 explain that mechanism.

With Level 3's problem statement defined, we can move on to Level 2's, which turns out to be quite similar. After all, from the top level's perspective, the only difference between them is that Level 3 supplies dynamic policies while Level 2 supplies static ones. However, we also require Level 2 to be available as a subroutine to Level 3. Therefore, we will have to adapt the inputs and outputs slightly to accomodate this additional role.

First, Level 2 needs to know whether the risk allocation it is looking for needs to yield a strongly or dynamically controllable STNU. Therefore, compared to Problem 5.2, Problem 5.3 takes an additional input specifying whether we want a static or dynamic policy. If it's the top level calling Level 2, then it will be static. Otherwise, it's Level 3 calling, requesting a dynamic policy. This parameter is ultimately passed down to Level 1.

In the dynamic case, we also allow Level 3 to include a starting set of linear constraints that the risk allocation has to satisfy. These linear constraints represent the facets of the

convex polytope obstacles that Level 3 has chosen to branch on. But they also include previously discovered *unconditional* linear conflict resolutions discovered in Level 2. The reason for this will be discussed in conjunction with the outputs below.

Problem 5.3 (Level 2 solving for a risk allocation)**.** *Given a pSTN, a reformulated chance constraint* rcc*, and any* a priori *linear constraints, find an assignment to the risk allocation variables that satisfies the* rcc and *yields a controllable STNU, either of the static or dynamic flavor, as specified by the caller. If we cannot generate a risk allocation, return a risk allocation conflict. In the dynamic case, if we can generate one, but encounter an SRNC, return that cycle's resolution instead of a policy.*

Inputs:

- *A pSTN* $\mathcal{N}^p$
- *A set of risk allocation variables* $\{l_1, u_1, \ldots, l_K, u_K\}$
- *A reformulated chance constraint* rcc *defined over those variables*
- *Whether we want a static or a dynamic policy*
- *In the dynamic case, an (optional) set of linear constraints over the variables*

Outputs:

- *A status that can be one of found, infeasible, or srnc*
- *If found or srnc, an assignment to the risk allocation variables*
- *If found or srnc, the implied STNU* $\mathcal{N}^u$
- *If found, a feasible policy* $\mathcal{P}$ *for* $\mathcal{N}^u$
- *If infeasible, a set of linear constraints that falsify* rcc
- *If srnc, a semi-reducible negative cycle from* $\mathcal{N}^u$*, and its disjunctive linear conflict resolution constraint*
- *In all cases, the entire set of learned linear conflict resolutions*

Compared to Level 3, Level 2's has three possible output categories, rather than the usual true-false or feasible-infeasible distinction. This third category pertains to the dynamic policy case, when it receives a semi-reducible negative cycle conflict from Level 1. We saw in Subsection 4.3.2 that DC conflict resolutions are disjunctive linear constraints, in contrast to the unconditionally linear resolutions for SC conflicts. Since Level 2 cannot process these disjunctive constraints, we have to send them back up to Level 3 for branching. For completeness, we also return the generated risk allocation and the resulting implied STNU that contains the offending DC conflict.

Besides this third case, though, Level 2's outputs are mostly identical as Level 3's in the affirmative and negative cases. When we find a satisfying risk allocation, we return that along with the implied STNU and a policy. Likewise, when we can't find such a risk allocation, we return a set of constraints which are known to block out the reformulated chance constraint region, thus forming a Level 2 risk allocation conflict. Those constraints represent learned resolutions to all the negative cycle conflicts that have been received from Level 1.

Internally, this infeasible set comes from Level 2's NLP solver not being able to find an assignment to the risk allocation variables. As with Level 3, we don't require this set to be *minimal*, because most NLP solvers do not provide *irreducible* infeasible subsets. In the future, it is certainly worth considering how to build that capability around such solvers. But generally, we will consider this infeasible set to be a subset of all the learned resolutions that were sent to the NLP solver.

The final difference with Level 3 is that since Level 2 does not process disjunctive constraints, it wouldn't make sense for its risk allocation conflicts to include such constraints. In other words, since Level 2 is expected to handle only linear conflict resolutions through its NLP solver, any conflict it generates must consist only of linear constraints. For static policies, this is always the case, since SC conflict resolutions are linear constraints. For dynamic policies, though, recall that *some* DC conflicts don't contain lowercase edges, and those also have unconditionally linear resolutions. Thus, Level 2 can actually resolve those conflicts itself, *until* it encounters one with a lowercase edge.

What this means is that when solving for dynamic policies and Level 2 returns with an

infeasible or *srnc* status, it needs to be prepared to be called again when Level 3 chooses a new combination of polytope facets to branch on. For this new call, Level 2 shouldn't forget about the unconditional conflict resolutions that it had learned in previous calls. Otherwise, it might generate an STNU that contains the same negative cycles, and be forced to rediscover them.

When designing these levels, though, we treat their invocations as stateless subroutines. Therefore, if Level 2 needs to keep a "running memory" of linear conflict resolutions across calls, it should return what it has learned to Level 3, and ask Level 3 to include them again in subsequent calls. This is the purpose of the last output of Level 2: a record of all the learned linear constraints, so that Level 3 can shuttle them back as part of Level 2's last input in the dynamic case. Level 2 also returns this record when it is successful, so that Level 3 has a complete "log" of all the SRNC conflicts that were discovered to reach the final solution.

Lastly, note that when Level 2 returns *infeasible*, the infeasible set is not the same as the learned resolutions. The former is a subset of all the linear constraints sent to the NLP solver, which includes linear facets selected by Level 3, all the unconditional linear resolutions learned in previous calls to Level 2, and those that have been learned during the current call. Any constraint in the last category may or may not be part of the infeasible set returned by the NLP solver.

Having completed Level 2's problem specification, we now turn to Level 1 for checking STNU controllability. This is the most straightforward level to specify, because by the time we call it, we have reduced the pSTN into an STNU, and scheduling STNUs is a solved problem. All we need in addition to the STNU is to specify whether we want a static or a dynamic policy. In turn, Level 1 will either provide such a policy, or return a negative cycle conflict.

In the case of a dynamic policy, the conflict is conditioned on valid extension subpaths for any lowercase edges in the negative cycle. Technically, these would be part of the SRNC "object" that is returned. For clarity, I list them explicitly in Problem 5.4.

Problem 5.4 (Level 1 checking STNU controllability)**.** *Given a grounded STNU, find a static or dynamic scheduling policy for it according to the caller's specification. If the*

STNU is uncontrollable, return a negative cycle conflict.

Inputs:

- *An STNU $\mathcal{N}^u$*
- *Whether we want a static or a dynamic policy*

Outputs:

- *Whether the STNU is controllable*
- *If yes, a scheduling policy $\mathcal{P}$*
- *If no, a negative cycle in $\mathcal{N}^u$'s distance graph*
- *If no and we are want a dynamic policy, a set of valid extension subpaths for the negative cycle's lowercase edges*

The remaining tasks of this chapter are to provide algorithms that implement each of these problem statements. While Section 5.3 onwards will focus on the conflict-directed structure within the numbered levels, the top level has a different role of reformulating the original problem into risk allocation. This is computationally distinct from the other levels, yet common to both our strategies for finding static and dynamic policies. Thus, we close this section with the implementation for the top level problem, given by Algorithm 5.1.

The top-level implementation can be understood in three sections: First, we set up the reformulation in lines 1–2 by creating the reformulated chance constraint, along with the risk allocation variables that it is expressed in. Then we call either Level 3 (line 4) or Level 2 (line 6) as our entry point, depending on whether we want a dynamic or static policy, respectively. Finally, we return the policy solution if successful (line 8), or we translate the infeasible outputs of Level 3 or 2 into a cc-pSTN conflict (lines 10–11).

As discussed earlier, the infeasible outputs from Levels 3 and 2 are a set of disjunctive linear or unconditional linear constraints, representing resolutions to discovered negative cycle conflicts coming from Level 1. Therefore, they can be mapped back into those negative cycles in the various implied STNUs encountered along the way. Since those STNUs are formed by applying risk allocations to the original pSTN $\mathcal{N}^p$, we can translate each edge in

Algorithm 5.1: Top-level problem reformulation and retrieval of solution from lower levels

Input: A pSTN $\mathcal{N}^p$
Input: A chance constraint with risk bound Δ
Input: Whether we want a static or a dynamic policy

1 ra-vars ⟵ Initialize risk allocation variables $l_1, u_1, \ldots, l_K, u_K$
2 rcc ⟵ Reformulate chance constraint in terms of risk allocation

3 **if** *dynamic policy requested* **then**
4 | *found*, rv-assign, $\mathcal{N}^u$, policy, feasible-LP, infeasible-constraints
| | ⟵ `CallLevel3(`$\mathcal{N}^p$, ra-vars, rcc`)`
5 **else**
6 | *found*, rv-assign, $\mathcal{N}^u$, policy, infeasible-constraints
| | ⟵ `CallLevel2(`$\mathcal{N}^p$, ra-vars, rcc, *static*`)`

7 **if** *found* **then**
8 | **return** *true,* policy, rv-assign
9 **else**
10 | cc-pSTN-conflict ⟵ Trace which components of the input cc-pSTN participate in infeasible-constraints
11 | **return** *false,* cc-pSTN-conflict, infeasible-constraints

those STNU cycles back into a component in $\mathcal{N}^p$. The cc-pSTN conflict is thus all those components collected, along with the original chance constraint, whose reformulated form was shown to be incompatible with all discovered conflict resolutions. This is what line 10 achieves.

Intuitively, this conflict tells the user that if they supply the same chance constraint risk bound, and try to enforce it against another pSTN with those same components, that cc-pSTN is guaranteed to be infeasible. The reason is that we just computed a series of risk allocations, which would be just as applicable to those shared components. From those risk allocations, we derived a set of necessary cycle resolution constraints, which we found to be ultimately incompatible with the risk bound. Because those cycles are covered by the shared pSTN components, they would be present in the STNUs implied by that series of risk allocations, and hence discoverable. Thus, we could "replay" the entire computation on the new pSTN, and reach the same conclusion.

5.3 Template for solving each layer

As presented at the beginning of Section 5.2, the motivation for separating out three layers of interacting problems was to illuminate the common master-and-subproblem relationship between Levels 3 and 2, and Levels 2 and 1. Therefore, when implementing each of those levels, it makes sense to follow common principles for observing that relationship. This section is thus dedicated to specifying those principles through a template algorithm. The numbered levels can then each be instantiated as versions of this template tailored to their particular problem specification.

Note that Level 1 can be considered an edge case, since it doesn't have a subproblem to call; it only reports to its Level 2 master. However, Levels 2 and 3 both have subproblems, and they report solutions or conflicts to the level above. Technically, Level 3 doesn't act as a subproblem to another master problem; the top level can be considered a shell that packages up the main functionality in Level 3 and below. However, our levels don't know where they're being called from, so Level 3 can be thought of as being a "subproblem" in the sense of returning conflicts to the end user, after undergoing a cosmetic repackaging.

Recall that in Section 4.2, we applied the principles of conflict-directed search specifically to our reformulated problem. This culminated in the algorithm flow illustrated in Figure 4-9, and written in Algorithm 4.1. Here, we take a step back and generalize slightly, so that we can then apply it to all three levels.

Figure 5-5 illustrates the nominal path of a solution through the algorithm for a level, *without* encountering any infeasibilities or conflicts. That is, imagine if we had a "certificate" for a correct solution handed down from an oracle. If we use that to guide our way through the algorithm, every check against the constraints will be satisfied, and we require no backtracking or looping.

The purpose of showing this nominal path is to remind ourselves that our conflict-directed approach separates out a subproblem from the entire problem that this level solves. The idea is to use a specialized solver for the subproblem that can return conflicts when infeasible. Abstractly, we can think of the entire problem as a collection of variables and constraints. Recall from the discussion surrounding Figure 4-6 that a subproblem is uniquely

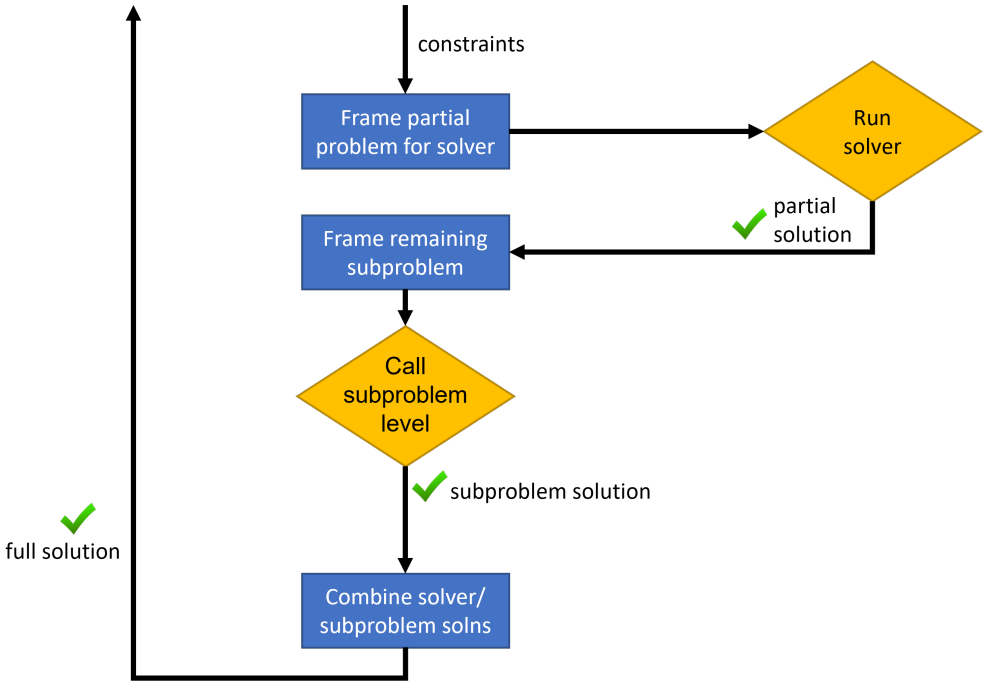

Figure 5-5: Given a problem, we first find a solution to a portion of its constraints, using an external solver. Combining this partial solution with the remaining constraints forms a subproblem, which is sent to the level below. If all goes well, the subproblem module returns a solution, which combines with our solver's partial solution to form a full solution to the problem *at this level*.

defined by assigning a subset of those variables, such that a subset of the constraints are satisfied. The subproblem then becomes specified by the remaining unassigned variables and the remaining constraints, projected as needed.

In Figure 5-5, we distinguish the tasks of assigning variables as yellow diamonds representing calls to encapsulated functionality. Pre-subproblem, we've isolated the "constraint solving logic" of this level in a solver module. Then we have the subproblem itself, for which we call an appropriate implementation. Around the yellow diamonds we have blue rectangles representing the "plumbing" required to frame the inputs to these external calls, as well as repackaging their outputs.

Since a full solution satisfies all the constraints, our algorithm could in principle generate it all in one pass without encountering any conflicts. That is, when running this level's solver logic on the non-subproblem constraints, there exists a valid solution to the non-subproblem variables that matches those given in the full solution. If the solver generates that partial solution, then for the subproblem, the remaining assignments in the full solution is a valid solution, and thus could be feasibly returned by the subproblem. Together, the solver's solution plus the subproblem's solution would form the full solution, and this level can thus return. So, the oracle's certificate would simply walk us through this problem decomposition, verifying along the way that all the original constraints are satisfied.

In real life, oracles don't exist, so we have to consider alternative possibilities and reasoning paths. Figure 5-6 amends the picture to include these. Namely, we ask what happens if the yellow diamonds are unable to find solutions for their respective constraints. As we discussed in Section 4.2, the first place this could happen is when the subproblem is infeasible. Here is where we exercise the central idea of extracting a conflict and turning its resolution expression into a learned constraint for the master problem at this level. Nominally, the solver can process this constraint, so we append it to the list of constraints the solver has to satisfy. Thus, this manifests the main loop of our conflict-directed approach.

The second place something could go wrong is with the current level's solver. Whatever reasoning it's performing, if it can't generate a partial solution, then the subsequent path to framing a subproblem and returning a solution is unreachable. We know this because in our framework, conflict resolutions are only ever accumulated, and constraints to the

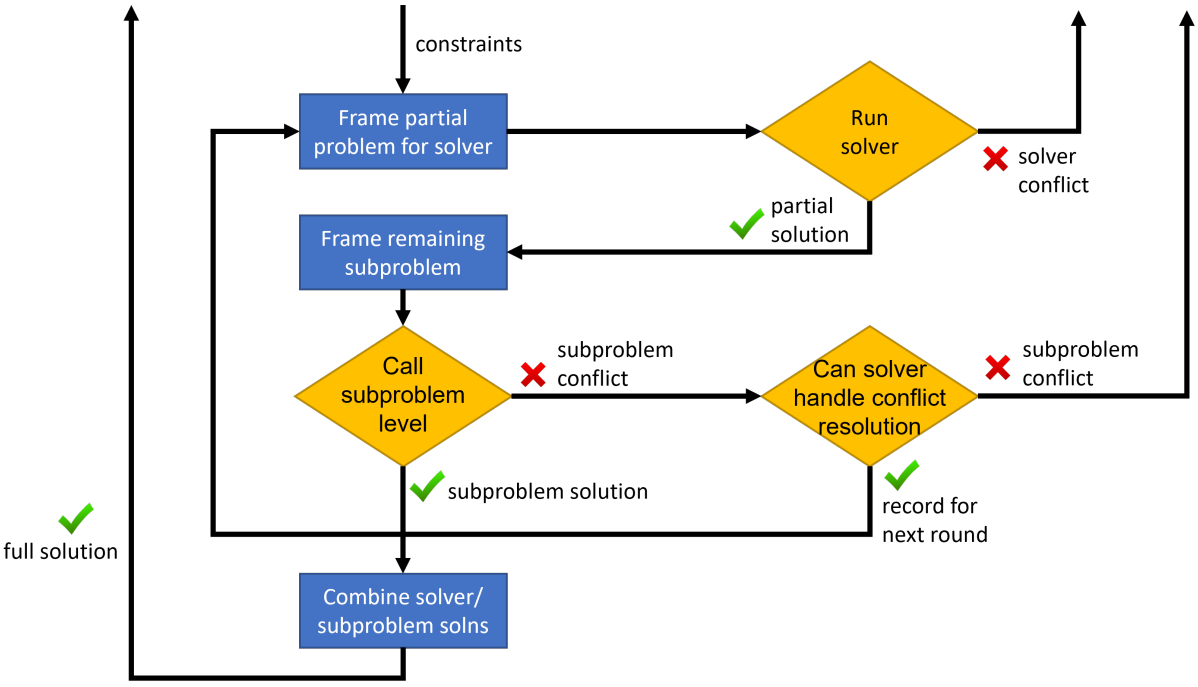

Figure 5-6: If the subproblem was unable to find a solution, then it returns a conflict that we then resolve via a learned constraint for the solver on the next round. In turn, this level itself may have conflicts to return to its caller. First, the solver may eventually face an infeasible set of constraints, and we return the solver's conflict (if available). Second, it may be the case that the *form* of a conflict resolution cannot be handled by the solver, during which we also kick the subproblem's conflict up to the caller.

solver are never discarded between rounds. Therefore, if the solver is given an infeasible set of constraints, adding to it will never make the solver return a solution in the future. At this point, we accept whatever conflict the solver returns, i.e., a subset of the constraints it last received, and return that the current level's own conflict. If the solver does not have a conflict-returning capability, we can always just return all of its constraints, or try to build our own analysis to extract a smaller infeasible set.

Finally, we revisit the spot where we processed the conflict from the subproblem level. We had assumed that our solver could handle the conflict resolution constraint, but as we've seen with STNU DC conflicts, their disjunctive resolutions cannot be handled by the NLP solver in Level 2. Thus, we require a check of the constraint form at this point, also represented by a yellow diamond. When we are faced with a learned constraint that the current level is not designed to handle, we can let a higher level handle it. In short, when the level below returns a conflict whose resolution we can't process, we pass that resolution up to the level above as a kind of conflict from the current level.

Algorithm 5.2 codifies the design of Figure 5-6 in pseudocode. Within the loop, the three decision points of the yellow diamonds are specified in lines 4–5, lines 8–9, and lines 12–13. In sequence, these run the solver at the master level, call the subsolver, and check compatibility of the conflict resolution. Each of these points has an exit route: We return a conflict when the master solver fails, we return a solution when the subproblem succeeds, and we pass up a conflict resolution that the master solver won't accept.

Structurally, this algorithm is very similar to Algorithm 4.1, which is specific to our overall risk allocation problem, but lacks details on handling SRNC conflicts. Here, we abstract away the problem specification into a collection of constraints, which we partition into $\mathcal{C}_1$ for the master solver and $\mathcal{C}_2$ for the subproblem. We also abstract away the learned conflict resolutions into a set $\mathcal{D}$. For brevity, we leave out the variables, which can be inferred from the constraints.

Since the subproblem won't handle them, we feed both $\mathcal{C}_1$ and $\mathcal{D}$ into our master solver. The partial solution we get out of it may include variables that appear in $\mathcal{C}_2$. Thus, we project $\mathcal{C}_2$ onto the subspace where the partial solution freezes certain variable assignments This yields $\overline{\mathcal{C}_2}$, which specifies the subproblem.

Algorithm 5.2: Template algorithm to be instantiated for solving each level

Input: A set of constraints $\mathcal{C}$
Output: Whether a solution was found
Output: If yes, an actual solution
Output: If no, a conflicting subset of $\mathcal{C}$

`// Initialize constraint sets`
$\mathcal{C}_1, \mathcal{C}_2 \longleftarrow$ Partition constraints according to solver's capabilities
$\mathcal{D} \longleftarrow$ Initialize empty set of learned constraints

while *true* **do**
  `// Query a solver on the constraints it can handle; return if infeasible`
  feasible, solver-sol, infeasible-subset $\longleftarrow$ `RunSolver(`$\mathcal{C}_1 \cup \mathcal{D}$`)`
  if *not feasible* **then**
    return *false,* infeasible-subset
  `// Formulate and solve a subproblem; return if feasible`
  $\overline{\mathcal{C}_2} \longleftarrow$ Project $\mathcal{C}_2$ onto solver-sol
  feasible, subproblem-sol, subproblem-conflict $\longleftarrow$ `CallSubproblem(`$\overline{\mathcal{C}_2}$`)`
  if *feasible* **then**
    full-solution $\longleftarrow$ Merge solver-sol and subproblem-sol
    return *true,* full-solution
  `// Translate subproblem conflict into a learned constraint; return if solver can't handle`
  conflict-resolution $\longleftarrow$ Lift subproblem-conflict into current level
  if *solver can handle* conflict-resolution **then**
    Collect conflict-resolution into $\mathcal{D}$
  else
    return *unhandled,* subproblem-conflict

The other difference between this algorithm and Algorithm 4.1 is that the latter didn't include a check for whether the solver could handle the conflict resolutio constraint. That was because it assumed it was solving the *entire* risk allocation problem, and thus had no master problem to report to. Here, this mechanism is necessary for Level 2 to send the disjunctive constraints that resolve SRNCs up to Level 3. In effect, we are breaking up the direct connection in Algorithm 4.1 from line 16 to line 7 into the if-else block in lines 13–16 of Algorithm 5.2. Whatever implied logic there was for Algorithm 4.1's solver to handle disjunctive constraints in line 7 is now delegated to Level 3.

5.4 Subsolver hierarchy for static policies

In this section, we compose algorithmic solutions for Level 2 and Level 1 to produce chance-constrained static scheduling policies. The end result is equivalent to Algorithm 4.1 with strong controllability on line 11. However, we rewrite it as two interacting layers, both of which are implemented as instantiations of the template Algorithm 5.2.

I begin with a series of diagrams that illustrate this interaction between Levels 2 and 1, as well as between the top level and Level 2. This gives us a birds-eye view of how the applicable problem statements in Section 5.2 are intended to compose into an architecture. Having given that oveview, I will then provide pseudocode for the two levels in a bottom-up manner: first Level 1 since it is the base case, and then Level 2.

Like we did with the template, let's start with the nominal path for generating a solution, but at the level of the entire architecture, rather than within a particular layer. Figure 5-7 shows this path, highlighted into three segments of different color. On the left side are blue rectangles representing the hierarchy of levels that get called. For Levels 2 and 1, I have separated out their solver modules to the right, previously represented in Figure 5-6 as the "Run Solver" diamond. This makes clear what "master" problem is being solved at each level, and what subproblem is passed to the level below.

First, in orange, a cc-pSTN in passed into the top level, then translated into the risk allocation problem via a reformulated chance constraint, which is passed into Level 2's NLP solver. In the next phase, in purple, the solver returns a risk allocation (i.e., an assignment

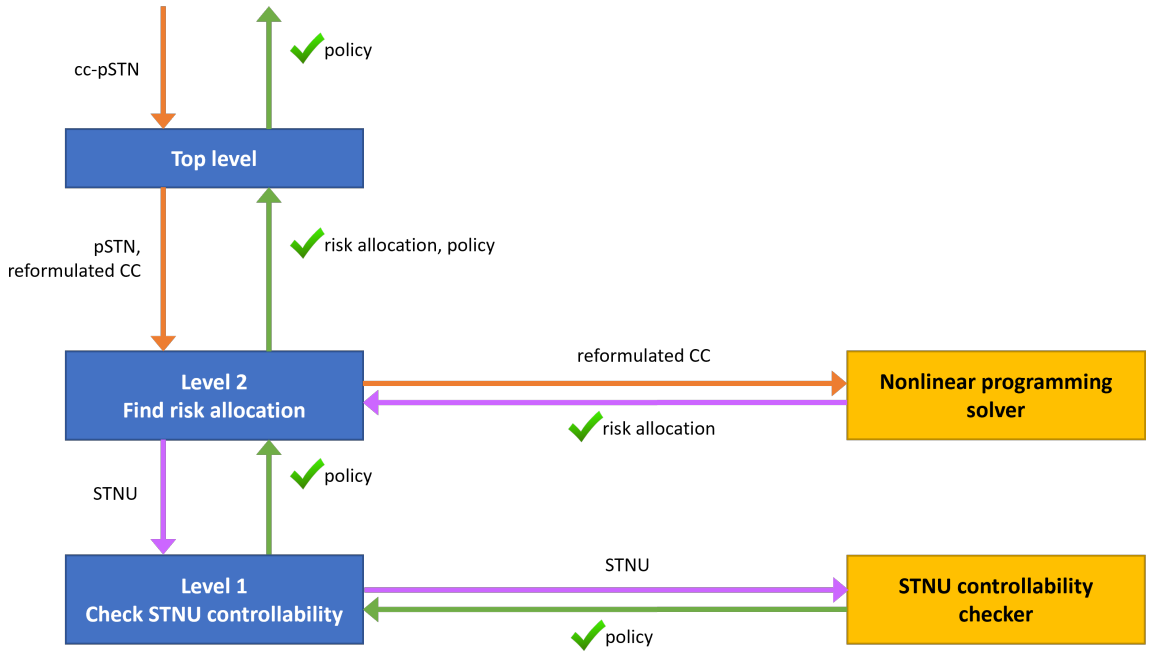

Figure 5-7: This is the nominal path for finding a static policy. From the top level, the reformulated chance constraint gets passed down to Level 2 into the NLP solver (orange path). Once that generates a risk allocation, the implied STNU gets passed down to Level 1 into the controllability checker (purple path). Finally, when the checker verifies controllability, a policy gets passed back up all the way to the top level.

to the risk allocation variables), from which the implied STNU is built and sent to Level 2's STNU strong controllability checker. Finally, in green, the checker finds us a static policy for the STNU, which gets tossed back up through the levels to the user.

With that nominal path laid out, we can start asking where negative results might be returned, and how the architecture should handle that. Figure 5-7 makes it easy to see that the *source* of all consequential decisions come from the yellow solver boxes to the right, which can either return a solution (feasible) or a conflict (infeasible). The blue boxes on the left just have to respond to those results by remapping them and shuttling them appropriately. Since the final policy solution comes from the lowest level, I will consider failures going backwards along the solution path, first when the STNU controllability checker returns infeasible, and then when the NLP solver fails.

Figure 5-8 shows the controllability checker returning a conflict when the STNU is not strongly controllable. As shown in Subsection 4.3.1, these are negative cycles in the STNU's distance graph. If we look back to Figure 5-6, the checker is Level 1's solver, so it's clear that Level 1 cannot handle this conflict. Therefore, it gets sent to Level 2, which derives a resolution to that conflict as a constraint in the risk allocation space.

Due to the unconditional nature of the negative cycle, the resolution is a linear constraint, and Level 2's NLP solver is able to handle this. Therefore, we add it to the nonlinear program, and resume the solution path starting from the Level 2 querying its solver. This establishes a potential "cycle of computation", bouncing between Level 2's and Level 1's respective solvers, with the levels themselves providing the communication in between. If at any point, Level 1's checker finds a policy, then we break out of the cycle and head to the top.

The other place in the architecture where a solver could fail is Level 2's NLP solver. It won't fail the first time it's called, because the only constraint given is the reformulated chance constraint. Unless the risk bound Δ is 0, it will always be possible to establish risk allocation cutoffs on the probabilistic durations to meet Δ. However, as conflicts come in from Level 1, and conflict resolution constraints get added, the remaining feasible space in the chance constraint region gets increasingly restricted. If at some point, the last constraint that was added blocks out a half volume that includes the entire remaining feasible space, then the solver returns in feasible.

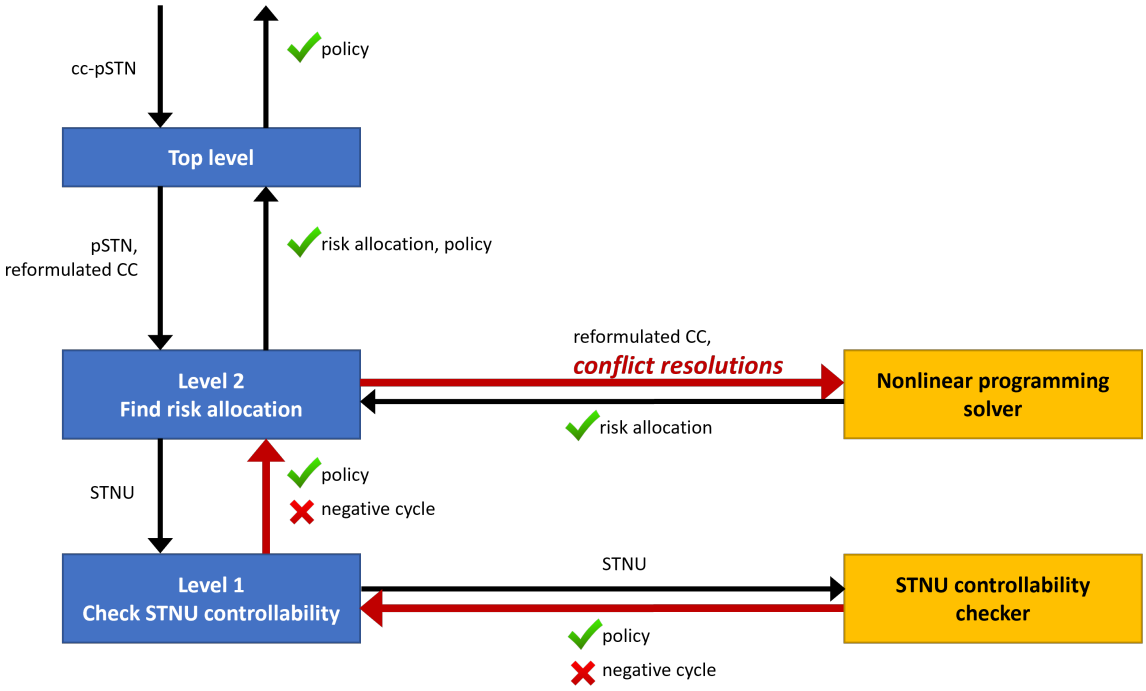

Figure 5-8: When the controllability checker fails, it returns a negative cycle conflict. This becomes the conflict for Level 1, which returns it to Level 2. Level 2 then has the responsibility of translating it into an expression in risk allocation space, and negating it so that it becomes a conflict resolution constraint. This is passed to the NLP solver to start the next round of risk allocation.

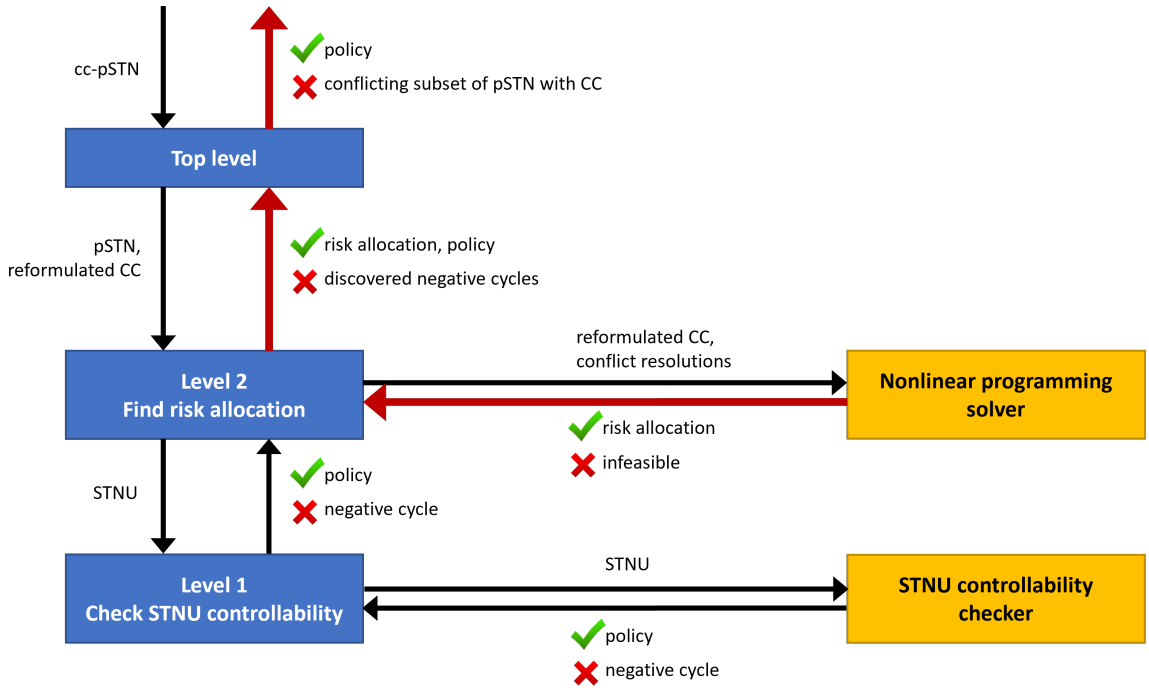

Figure 5-9: When the NLP solver fails, that means the learned conflict resolutions altogether are incompatible with the reformulated chance constraint. Thus, Level 2 returns the set of negative cycle conflicts collected from Level 1 as its own conflict to the top level. The top level then traces these cycles' edges back to the components of the original pSTN, and together with the original chance constraint, that forms the entire algorithm's "conflict" for the cc-pSTN.

According to the template in Figure 5-6, this is a Level 2 conflict. Thus, Figure 5-9 shows it getting returned to the top level, which then translates it into a cc-pSTN conflict. For simplicity, we show only the simplest possible conflict extraction strategy for Level 2, which is to treat the entire NLP as the conflict. It may be possible to trim down the conflict to an irreducible infeasible set, but that is out of scope here.

By the argument above for when Level 2's solver would return infeasible, we know that the chance constraint is always involved in the conflict, and whoever called Level 2 already knows about the chance constraint, since they were the one who passed it in. Therefore, the only real new information is the collection of all the learned conflict resolution constraints. In turn, these come directly from the negative cycles that were discovered by Level 1. So even though officially and in the pseudocode below, Level 2 is returning a set of linear constraints that are infeasible with the chance constraint as a conflict, we can intuitively

think of it as returning the set of discovered negative cycles that were collected from Level 1 (or a subset thereof). In practice, if Level 2 returns the negative cycles, rather than their linear resolution constraints, then that saves the top level one step of "backward lookup" when mapping the Level 2 conflict into the cc-pSTN's components.

One final note: It is technically possible for the chance constraint to *not* be involved in a Level 2 conflict, if Level 1 returns a negative cycle that includes no edges from an uncontrollable duration. That just means there is an STN-style negative cycle, where all edges are constant, and will thus remain unchanged for any risk allocation.[4] In this case, that conflict is directly present in the original pSTN as well, and does not involve the chance constraint.

The resolution constraint for such a conflict would thus be a false statement, namely, that a sum we know to be negative is non-negative. Therefore, Level 2 can return this directly to the top level as its conflict without having to send it through the NLP solver. For conciseness, I did not display this edge case in the previous figures, nor will I in the following pseudocode, but it is simple enough to check for it. In any case, if we feed the false statement in to the NLP solver as in Figure 5-8, then the solver would return infeasible, so the final output would still be correct.

Figure 5-9 thus completes the architecture for generating static chance-constraint policies. For the remainder of this section, I provide the pseudocode implementations for Level 1 and Level 2, referencing the template in Algorithm 5.2.

Level 1 is rather straightforward. After checking controllability of the input STNU, there is no further Level 0 subproblem, and thus no conflicts from below to process. Therefore, as Algorithm 5.3 shows, there is no need to loop either. Level 1 thus functionally becomes a wrapper around an actual STNU controllability-checking algorithm.

Although this section focuses on strong controllability, Level 1's implementation can

[4]Even though the conflict is present, that does not guarantee the controllability checker will return that specific one. If there are other negative cycles in the distance graph, then due to the nature of the underlying SSSP Bellman-Ford algorithm, any one of those negative cycles could be returned, and *only* one would be. However, even though there's no guarantee that the constant-weight cycle would be discovered in the first round, it will *eventually* be discovered, provided that the accumulated conflict resolutions did not make Level 2's NLP risk allocation problem infeasible. This is because there are only a finite number of such cycles that can be discovered, so at some point, we must be forced have resolved all other cycles, and have no choice but to discover this one.

Algorithm 5.3: Instantiation of template for Level 1: checking controllability

Input: An STNU $\mathcal{N}^u$
Input: Whether we want a static or dynamic policy

```
// Query an STNU controllability-checking algorithm; return if
   uncontrollable
1 controllable, policy ⟵ CheckControllability(N^u, policy-type)
2 if not controllable then
3 |   return false, neg-cycle

// There is no Level 0 subproblem to call; return true
4 return true, policy
```

be easily adapted for dynamic controllability by simply taking in a parameter indicating the desired type of scheduling policy. Its solver module can then use this parameter to switch between algorithms that check for strong or dynamic controllability. In both cases, the solver's conflicts are negative cycles in the STNU distance graph. And although earlier, for clarity, we had specified in Level 1's Problem 5.4 that DC conflicts are accompanied by an additional output containing extension paths, it makes the pseudocode cleaner to consider the extension information as part of the SRNC negative cycle. Hence, line 3 is identical for the static and dynamic cases.

Level 2's implementation for producing static policies is given by Algorithm 5.4, which mirrors the template algorithm's structure closely. Namely, the body of the while loop consists of three major steps, corresponding to the three diamonds in Figure 5-6. First, we use an NLP solver to solve the master problem. Then, we formulate the STNU subproblem, and call on Level 1 to solve it.

The main difference is that in the third step, we don't need to check whether the conflict resolution is compatible with the NLP solver We already know that the resolutions are linear constraints, so we add it directly to the set of learned constraints. This set is initialized before the while loop, and corresponds to the set $\mathcal{D}$ in the template.

Note that during initialization, partitioning of the input constraints is not required. This is because the $\mathcal{C}_2$ constraints which would have expressed STNU strong controllability are implied when we call Level 1. In other words, we are swapping out a generic subproblem solver for a specialized solver STNU controllability. And the generic projection of $\mathcal{C}_2$ into

Algorithm 5.4: Instantiation of template for Level 2: finding a risk allocation

Input: A pSTN $\mathcal{N}^p$
Input: A set of risk allocation variables ra-vars
Input: A reformulated chance constraint rcc
Input: Policy-type = *static*

```
// Initialize set of learned constraints
```
learned-constraints $\longleftarrow \{\}$

while *true* **do**
```
    // Query an NLP solver to generate a risk allocation; return
       if infeasible
```
    feasible, rv-assign, infeasible-set
    $\longleftarrow$ `CallNonlinearSolver`(ra-vars, rcc, learned-constraints)
    if *not feasible* **then**
        return *false,* infeasible-set

```
    // Formulate implied STNU for Level 1; return if
       controllable
```
    $\mathcal{N}^u \longleftarrow$ Project rv-assign onto $\mathcal{N}^p$
    controllable, policy, neg-cycle $\longleftarrow$ `CallLevel1`($\mathcal{N}^u$, *static*)
    if *controllable* **then**
        return *true,* rv-assign, $\mathcal{N}^u$, policy

```
    // Translate Level 1 conflict into learned constraint
```
    neg-cycle-RA $\longleftarrow$ Express neg-cycle in terms of ra-vars
    neg-cycle-RA-res $\longleftarrow$ Negate neg-cycle-RA to form resolution constraint
    Collect neg-cycle-RA-res into learned-constraints

$\overline{C_2}$ is represented by transforming the pSTN $\mathcal{N}^p$ into the implied STNU $\mathcal{N}^u$. Thus, we obtain the benefit of not only the specialized solver's efficiency, but also its conciseness of inputs.

Together with Algorithm 5.1 for the top level, Algorithms 5.4 and 5.3 complete our solution for generating static policies. They are functionally equivalent to running Algorithm 4.1 with an NLP solver. As the next section shows, though, adapting the architecture to supply dynamic policies requires only a minor change to Level 2's implementation plus an implementation for Level 3.

5.5 Subsolver hierarchy for dynamic policies

Like in the previous section, I begin by presenting how the different levels communicate within the architecture for generating dynamic policies. A third level will be inserted between the top level and Level 2, while the new type of SRNC conflicts from Level 1 will increase the complexity of communication up and down the chain. In terms of pseudocode, only a small change is needed to Level 2's implementation, and Level 3 will handle the rest.

Figure 5-10 presents the nominal path for generating a solution. It is much the same as Figure 5-7 in the static case, except we've inserted a Level 3. Recall that Level 3's job is to help us branch on disjunctive SRNC conflict resolutions. However, we won't have any until such a conflict is returned from Level 1, which subsequent figures will illustrate. Therefore, on the first pass through Level 3, its "master problem" of finding a leaf in its search tree is empty. This means Level 3 functions only a passthrough at this point in the solution process. The conceptual distinction of segments along the solution path as problem reformulation, risk allocation to STNU, and policy solution still holds.

Now we start considering possible solver failures along the way, starting from the bottom. Figure 5-11 considers negative cycles from Level 1 being translated into learned constraints for Level 2's solver, just as Figure 5-8 for strong controllability did. For this red path to hold, though, the conflict resolutions that Level 2 sends to its NLP solver must be *unconditional linear* constraints. We know that certain SRNCs may have such resolutions, namely, if they don't contain lowercase edges that require extension paths to reduce them

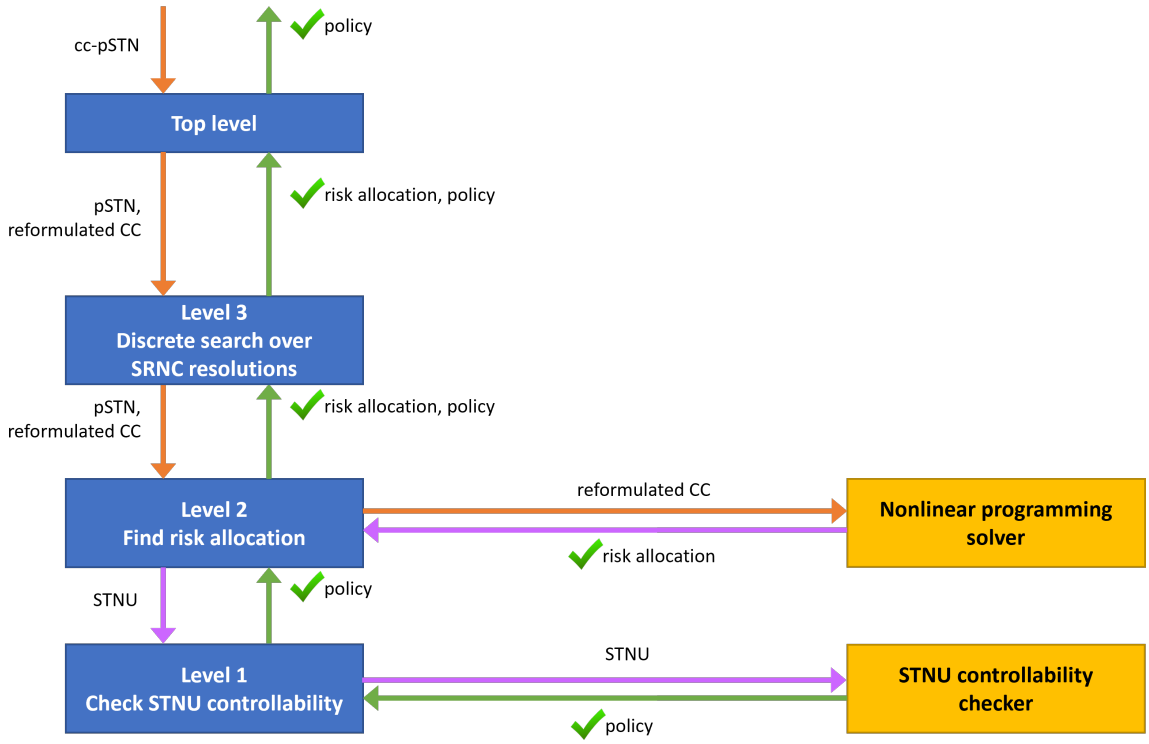

Figure 5-10: Just like in the static case, the nominal path for finding a dynamic policy goes through two stages of external solvers before sending a policy solution all the way backc up. However, a third level is present to handle disjunctive conflict resolution constraints as they come up. At first, this level acts just as a pass-through.

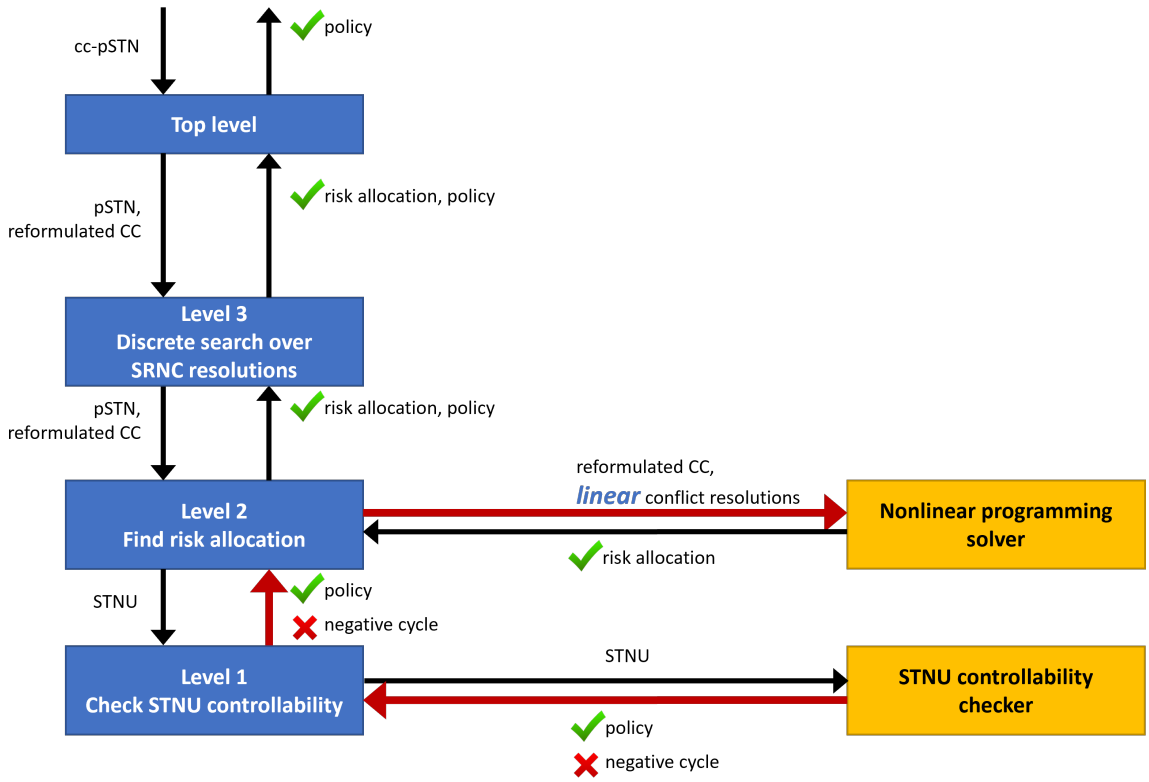

Figure 5-11: When the controllability checker returns negative cycle conflicts, as long as they don't contain any lowercase edges, they can be transformed into linear conflict resolutions. Hence, these can be passed directly to the NLP solver in Level 2, just like in the static case.

away. Therefore, Level 2 should check for this condition, and if true, allow the resolutions to proceed to the solver.

It could also be the case that lowercase edges do exist in the SRNC, but their extension paths do not contain any edges corresponding to an uncontrollable duration's bound in the STNU. Those paths' weight expressions, when mapped back into the risk allocation space, are purely constants, just like the STN-style conflicts mentioned previously on page 193. Thus, it is impossible to resolve the SRNC conflict by choosing to make one of those extension paths have non-negative weight. The only remaining choice of resolution is to make the entire cycle's weight non-negative, which Level 2's solver is happy to consider. Again, this is not shown explicitly in the figures or in the pseudocode, but its implementation is encouraged.

The real question for handling dynamic controllability conflicts is when the extension

paths *do* offer viable resolutions, via alternative conditions in the risk allocation space. Figure 5-12 highlights the communication in the architecture that takes place, as well as bringing in the specialized "solver" for Level 3. For convenience, I refer to an SRNC in Level 1 that would generate a disjunctive resolution in Level 2 as simply *the* discovered SRNC. The first key difference is that Level 2 can't handle that resolution, so it kicks it back up to Level 3. In terms of the figure, recall that although Level 2 technically sends up the disjunctive resolution constraint, I show it as sending the SRNC itself, to remind us of what the constraint represents.

Notice that in addition to sending the SRNC to Level 3, we also send all the negative cycles with non-disjunctive resolutions that Level 2 had discovered prior to the SRNC. The reason for this was given when discussing Level 2's problem statement in Problem 5.3. Namely, when Level 2 gets called again by Level 3, it shouldn't relearn from scratch any previous conflict resolutions that it can handle. But Level 2 is called with a fresh set of inputs each time, so these other cycles should be "saved" and "restored" between invocations. Level 2 could have also been made to maintain its internal state, but it's conceptually cleaner to present its pseudocode as stateless. Whether to implement it as stateless or not in practice is a choice we leave to the reader.

Once Level 3 receives the disjunctive constraint, its job is to frame a series of subproblems for Level 2 that involve only unconditional linear constraints. Internally, it abstracts away each disjunctive linear constraint into a discrete variable whose domain size is exactly the number of disjuncts. Selecting a particular disjunct to be included in what's sent down to Level 2 thus maps to assigning a particular value to the corresponding discrete variable.

The task of assigning all such variables becomes Level 3's "master problem", and the corresponding "solver" can be thought of as enumerating a next full assignment to those variables. I say "enumerating" because we want to try all combinations of one linear constraint from each disjunction, so Level 3's master solver is performing combinatorial search. Between calls to Level 2, we want to avoid enumerating the same combination, which would result in the same conflicts being returned (assuming Level 2's and Level 1's solvers run deterministically).

Once we have an assignment to all discrete variables, Level 3 maps that back into the

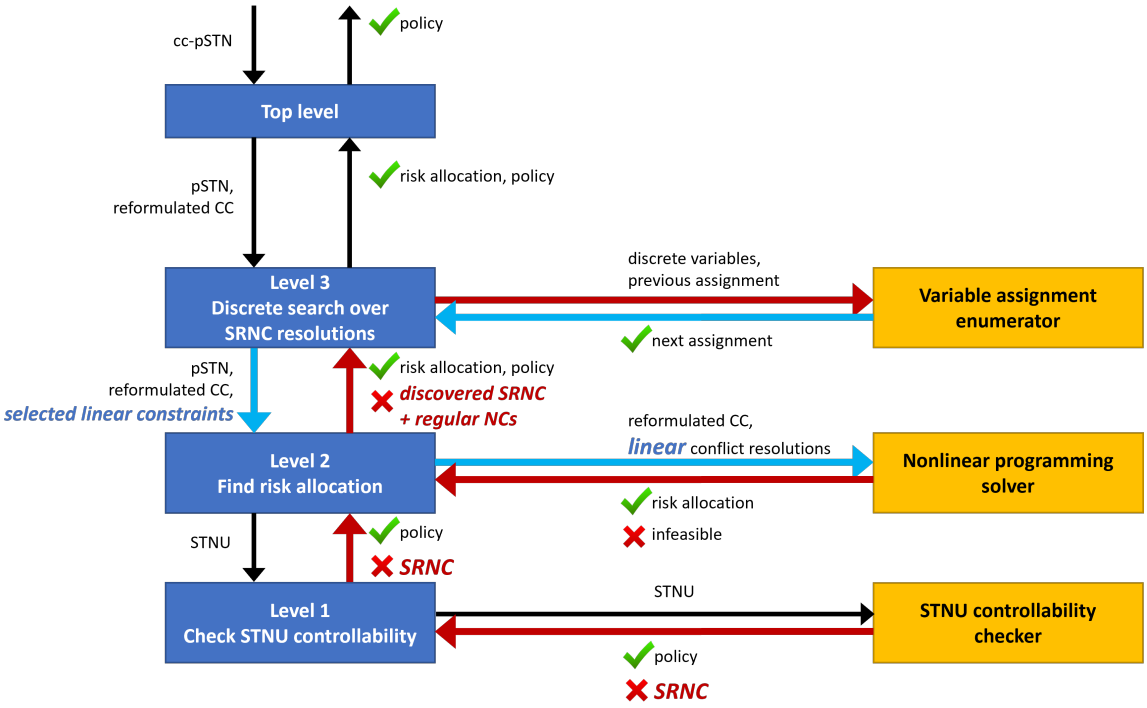

Figure 5-12: If we discover an SRNC with lowercase edges, then Level 2 has to send it up to Level 3 for branching. Level 3 maps the SRNC into a new discrete variable, with a domain value for each "facet" of its convex polytope obstacle. Branching in Level 3 is then performed by a variable assignment enumerator, which traverses the search tree and stops at its leaves. Each leaf gets translated back into an LP, which is sent back down to Level 2 to "jump start" the NLP solver (light blue path). Since Level 2 does not remember its state from its previous invocation, it's necessary for it to report the "unconditional" negative cycles as well, so they can be restored in the NLP on subsequent calls to Level 2. Finally, we note that when the NLP itself returns infeasible, we have the option of handling this "Level 2 conflict" using the same mechanism.

selected disjuncts, which are linear constraints. Those are then combined with all the learned linear constraints from previous calls to Level 2, and then sent back down to initialize the nonlinear program that forms Level 2's master problem. This is shown as the light blue path, where the output from Level 3's enumerator directly specifies the subsequent input to Level 2's solver. In effect, the unconditional linear constraints from Level 2 can be viewed as singleton disjunctions that get returned to Level 3. At that point, it is unnecessary to perform explicit branching on them, so they simply accumulate within Level 3, and get sent back verbatim to Level 2 on each call.

The last point to make about Figure 5-12 is that failures from Level 2's NLP solver are included in the red path as well. We know from Level 2's Problem 5.3 that the conflict when a risk allocation can no longer be generated is a subset of the linear constraints that were input to the NLP solver. That is, the *form* of the conflict is a conjunction of linear constraints in terms of the risk allocation variables.

This is exactly the same form as the conflict expression for an SRNC lifted into the risk allocation space! Therefore, its *resolution* is also a disjunctive linear constraint, which can be handled by the same branching mechanism in Level 3. So the interpretation of the red arrow going out of Level 2's solver in Figure 5-12 is that when infeasible, the conflict it returns follows the same path upstream into Level 3 for resolution. Even though the conflict is not technically an SRNC, it follows the same form algebraically.

To complete the diagram, Figure 5-13 shows what happens when Level 3's "solver" returns infeasible. Recall that for static policies, when Level 2's solver failed, the entire solution process stopped, returning back to the top level, where a cc-pSTN conflict was generated. Now that Level 3 is the highest numbered level, the same principles applies. The question is what condition causes Level 3 to fail, and what is the conflict to be returned.

Since Level 3's solver is performing combinatorial search by *enumerating* the remaining assignments, it fails when it has exhausted the space of assignments to its discrete variables. Typically in combinatorial search, there would be explicit constraints over those variables, leading to certain combinations of their assignments being infeasible. And if all combinations were infeasible, then one might be able to extract a subset of the constraints that mutually block out the solution space.

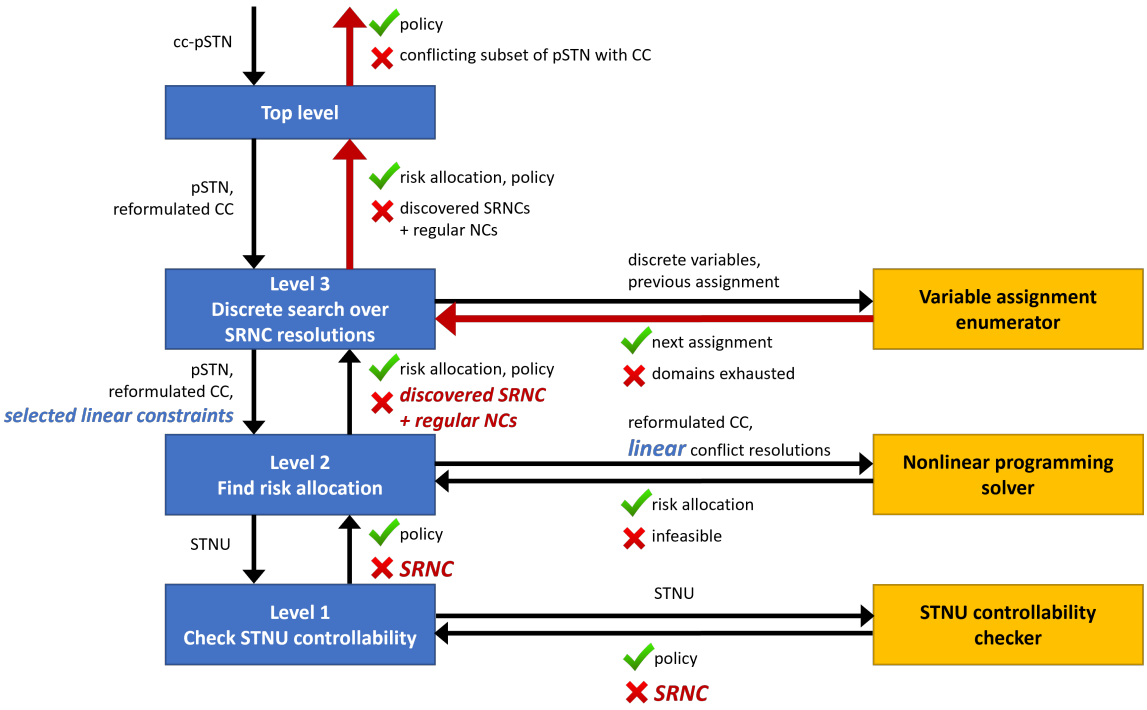

Figure 5-13: Our overall algorithm fails only when Level 3 has exhausted its search tree without finding a satisfying dynamic policy. We now know, just like in the static case, that all our discovered negative cycles cannot be resolved together with satisfying the reformulated chance constraint. Thus, we make sure all those cycles make their way to the top level, where they get traced back to the pSTN's components.

In our case, our discrete variables are "discovered" along the way, corresponding to each SRNC (or Level 2 infeasibility) that was sent up to Level 3. Thus, we have no explicit constraints over them; they are all implied by the risk allocation conditions in Level 2 and the STNU controllability conditions in Level 1. Nominally, what we can say then is that when Level 3 has exhausted all assignments, we have tried all possible combinations of resolutions to all known SRNCs, and none of them yielded a feasible risk allocation and dynamically controllable STNU. Therefore, this collection of SRNCs should be comprise Level 3's conflict report to the top level.

In addition to the SRNCs with disjunctive resolutions, we also have resolution constraints for negative cycles that have only one possible linear constraint resolution in the risk allocation space. As mentioned before, these can be considered as nodes in Level 3's search tree with singleton branching factors. Therefore, they are also collected into Level 3's conflict.

Recall that Level 3 is ultimately responsible for solving the reformulated risk allocation problem, according to Problem 5.2. Therefore, it is no surprise that Level 3's conflicts mirror Level 2's, but with the addition of SRNC conflicts, which form convex polytope obstacles rather than half-volumes in the risk allocation space. This was the intuition expressed in Section 5.1 on page 169. It also hearkens back to Algorithm 4.1, where on line 7 the abstract solver returns an infeasible subset of resolutions to all known conflicts. In our case, we have split that solver into Level 3's enumerator and Level 2's NLP solver.

To summarize how this layered architecture works, each level decomposes the problem given to it into its own master problem and a subproblem for the level below. This results in a computational path where the full solution gets partially built at each level's solver. When a solver fails, though, its conflict gets translated into a resolution constraint for the level above. In turn, that level decides whether its solver can handle the resolution, or whether it needs to be kicked up yet another level. The entire computational path is thus an iterative parley between Levels 2 and 1, recursively nested within iterations that reach into Level 3. Each time Level 2's solver is called, it always receives at least as many constraints as it did last time, corresponding to the number of known conflicts so far.

Algorithm 5.5 adapts Level 2's implementation from Algorithm 5.4 to produce dynamic

policies instead of static ones. The key difference is at the end of the while loop, where we append the negative cycle (SRNC) resolution to Level 2's constraint program only if it's an unconditional linear constraint. Otherwise, we know it's a disjunctive linear constraint, and we return it to Level 3 with an *srnc* status.

Algorithm 5.5: Instantiation of template for Level 2 in the dynamic case: finding a risk allocation

Input: A pSTN $\mathcal{N}^p$
Input: A set of risk allocation variables ra-vars
Input: A reformulated chance constraint rcc
Input: Policy-type = *dynamic*
Input: A set of linear constraints given-constraints

`// Initialize set of learned constraints`
1 learned-constraints $\longleftarrow \{\}$

2 **while** *true* **do**
  `// Query an NLP solver to generate a risk allocation; return if infeasible`
3 *feasible*, rv-assign, infeasible-set $\longleftarrow$ `CallNonlinearSolver`(ra-vars, rcc, given-constraints $\cup$ learned-constraints)
4 **if** *not feasible* **then**
5 **return** *infeasible,* infeasible-set, learned-constraints
  `// Formulate implied STNU for Level 1; return if controllable`
6 $\mathcal{N}^u \longleftarrow$ Project rv-assign onto $\mathcal{N}^p$
7 *controllable*, policy, neg-cycle $\longleftarrow$ `CallLevel1`($\mathcal{N}^u$, *dynamic*)
8 **if** *controllable* **then**
9 **return** *found,* rv-assign, $\mathcal{N}^u$, policy, learned-constraints
  `// Translate Level 1 conflict into learned constraint; return if disjunctive`
10 neg-cycle-RA $\longleftarrow$ Express neg-cycle in terms of ra-vars
11 neg-cycle-RA-res $\longleftarrow$ Negate neg-cycle-RA to form resolution constraint
12 **if** neg-cycle-RA-res *is unconditionally linear* **then**
13 Collect neg-cycle-RA-res into learned-constraints
14 **else**
15 **return** *srnc,* rv-assign, $\mathcal{N}^u$, neg-cycle-RA-res, learned-constraints

There are a few additional minor differences: We pass in the *dynamic* keyword for the desired type of policy, so it can be appropriately passed down to Level 1 on line 7. We also

accept an input list of given linear constraints, selected by Level 3, and merge those with the learned constraints when calling the NLP solver on line 3. These differences are minor enough that Algorithm 5.5 can be considered suitable for producing static policies, and thus a complete solution to Problem 5.3. We just have to call it with the policy type *static*, plus an empty list for the given constraints. The if-else block on lines 12–15 is guaranteed to fall into the if case, and thus the algorithm works identically to Algorithm 5.4.

The final piece to specify is the algorithm for Level 3's Problem 5.2, and this is given by Algorithm 5.6. Like with Level 2, this algorithm follows the structure of the template Algorithm 5.2. We begin by initializing the sets of learned conflict resolutions, followed by a while loop whose body consists of three decision blocks in sequence. For the learned constraints, we distinguish between the disjunctive resolutions that Level 2 can't handle, and the unconditional linear constraints that Level 2 discovered before encountering a disjunctive one. We then maintain a mapping from the disjunctive constraints to a set of search variables, which Level 3's enumerator acts on.

The enumerator needs the current (i.e., previous) assignment to the search variables to know what options have been eliminated and hence what is the next full assignment. Internally, one typically establishes an ordering to the domain values of each variable, so traversing the nodes of the search tree occurs in a well-defined order. We get a next assignment from the enumerator, and that becomes our new current assignment. If no such assignment could be found, due to the search space being exhausted, then as discussed previously, we return on line 8 all the learned STNU negative cycle resolutions, which is the union of the disjunctive and unconditional constraints.

The second block "projects" the discrete search variable assignment into a set of selected disjuncts, one from each constraint in disjunctions. These are combined with all the known unconditional constraints, and sent down to Level 2 as the subproblem. No matter the result of Level 2, the first thing we do when it returns is to collect any new unconditional linear constraints it learned. Then, if Level 2 was successfull, we have a policy solution to return. Also, according to Problem 5.2, we include the full list of selected disjuncts plus unconditional constraints as proof of a feasible risk allocation program (with the reformulated chance constraint implied) solved by Level 2's NLP solver.

Algorithm 5.6: Instantiation of template for Level 3: branching on linear inequality disjuncts

Input: A pSTN $\mathcal{N}^p$
Input: A set of risk allocation variables ra-vars
Input: A reformulated chance constraint rcc

```
   // Initialize different sets of learned constraints
1  disjunctions ⟵ {}
2  unconditionals ⟵ {}
3  search-vars ⟵ {}
4  sv-assign ⟵ {}

5  while true do
       // Query an enumerator for next search variable assignment;
          return when exhausted
6      exists, sv-assign-next
          ⟵ CallVariableEnumerator(search-vars, sv-assign)
7      if not exists then
8          return false, disjunctions ∪ unconditionals
9      sv-assign ⟵ sv-assign-next

       // Formulate risk allocation subproblem for Level 2 based on
          selected constraints; collect learned constraints, and
          return if feasible
10     selected ⟵ Map sv-assign into selected linear constraints
11     status, rv-assign, 𝒩^p, policy, infeasible-set, disjunctive-constraint,
        linear-constraints
          ⟵ CallLevel2(𝒩^p, ra-vars, rcc, selected ∪ unconditionals)
12     Collect linear-constraints into unconditionals
13     if status = found then
14         return true, rv-assign, 𝒩^u, policy, selected ∪ unconditionals

       // Translate Level 2 disjunctive conflict resolution into
          discrete search variable
15     if status = infeasible then
           // Handling risk allocation conflicts is descoped
16     else if status = srnc then
17         Collect disjunctive-constraint into disjunctions
18         new-svar ← Map disjunctive-constraint into a new discrete search variable
19         Collect new-svar into search-vars
```

Finally, we reach the last decision block, which determines how to handle conflicts returned from Level 2. Since this is the highest numbered level, we must process any conflict that Level 2 returns, instead of kicking it up to the top level. However, Level 2's problem statement allows for *two* types of failures, either *infeasible* where the NLP solver can't find a risk allocation, or *srnc* where it can't process a disjunctive SRNC resolution.

When discussing Figure 5-12, we had shown that both these types of conflicts share the same algebraic structure in the risk allocation space. Therefore, in principle, we should be able to handle them identically. We already know how to handle the SRNC conflicts, which form convex polytope obstacles. The intuition given in Section 5.1, and the whole point of Level 3, was to perform branching on those obstacles' facets.

A key observation, though, is that the number of such obstacles increases each time Level 2 returns with an *srnc* status. The number of obstacles corresponds directly with the depth of the search tree, which lines 17–19 are exactly doing by constructing a new search variable with the appropriate domain size. When we then return to the top of the while loop and call the enumerator on line 6, we will be *extending* the current branch we're on in the search tree by branching on that SRNC's disjuncts, rather than backtracking on the disjuncts for the already known SRNCs.

This is expected behavior for dealing with SRNCs as they get discovered. However, if we apply the same behavior to when Level 2 returns *infeasible*, then we will *also* continue to branch rather than backtrack. Our traversal of the search tree would end up being a single branch going all the way down, and thus miss out on the opportunity to resolve our known SRNCs via their other branches.

To avoid this outcome, our Level 3 enumerator needs to deduce that a node in its search tree is a dead end, and thus it should backtrack. This would require building in some sort of inference or propagation procedure, interleaved with the enumerator's search. For this thesis, we consider that out of scope, and thus declare that *infeasible* results from Level 2 *are* the dead ends in the search tree. This corresponds to doing nothing when line 15 is satisfied, thus allowing the next call to the enumerator to backtrack.

For completeness, though, we illustrate how we could collect a new convex obstacle as lines 17–19 do while still achieve backtracking. Figure 5-14 depicts a hypothetical situation

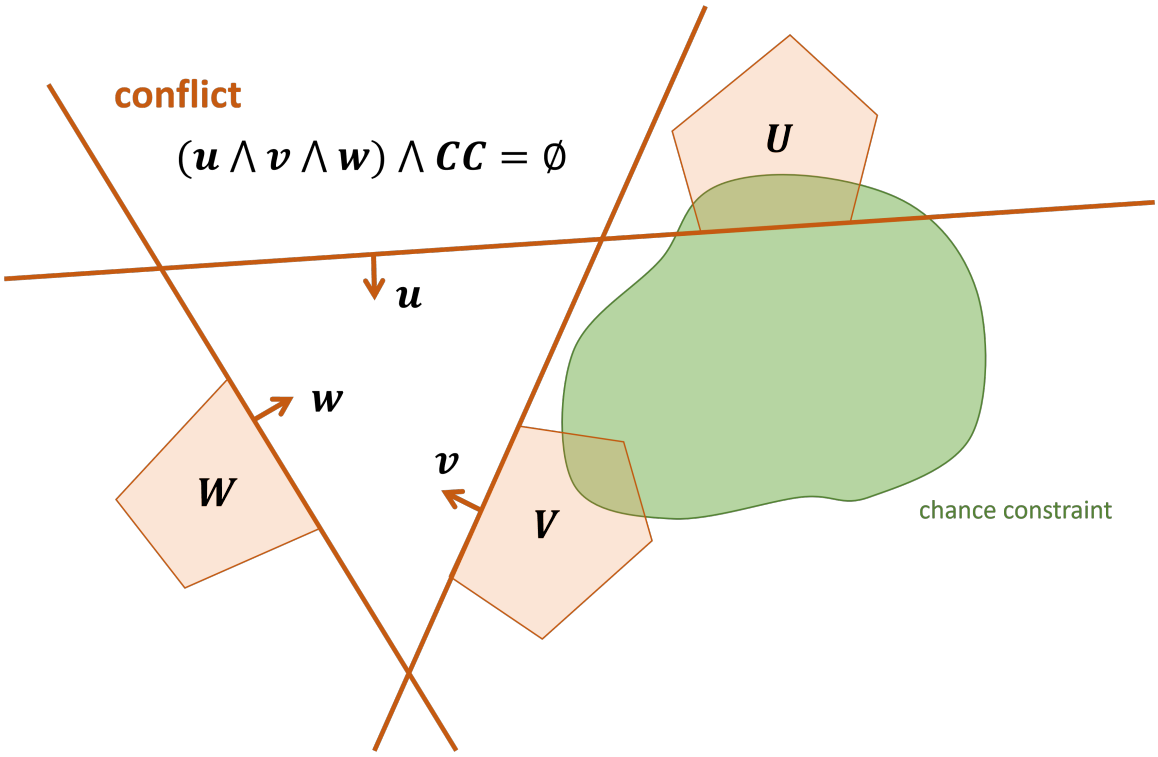

Figure 5-14: When Level 2 returns *infeasible*, it is because a subset of the selected facets ask us to find a risk allocation in a region that has no intersection with the chance constraint feasible region.

where we have three SRNC obstacles, U, V, and W, and we've branched on facets u, v, and w from each of them, respectively. If Level 2 returns *infeasible*, it is because staying on the "outside" of each of these three facets forms a region that lies entirely outside of the chance constraint. Therefore, the Level 2 conflict is the conjunction of these facets plus the chance constraint.

We can't change the chance constraint, but we could try alternative facets of the existing obstacles to help us stay out of this region. Therefore, the resolution of this conflict is a disjunctive linear constraint $\neg u \vee \neg v \vee \neg w$ telling us to stay "inside" at least one of the facets from the conflict. If the enumerator had a logical inference procedure built in, then it would recognize that any future branch containing both u and $\neg u$ is a dead end, and thus backtrack. Better yet, if it could perform unit propagation, then once it selected u on a future branch, it would remove $\neg u$ from this new constraint. Thus, by the time we branched on this constraint, and assuming $\neg v$ and $\neg w$ hadn't also been eliminated, we would only

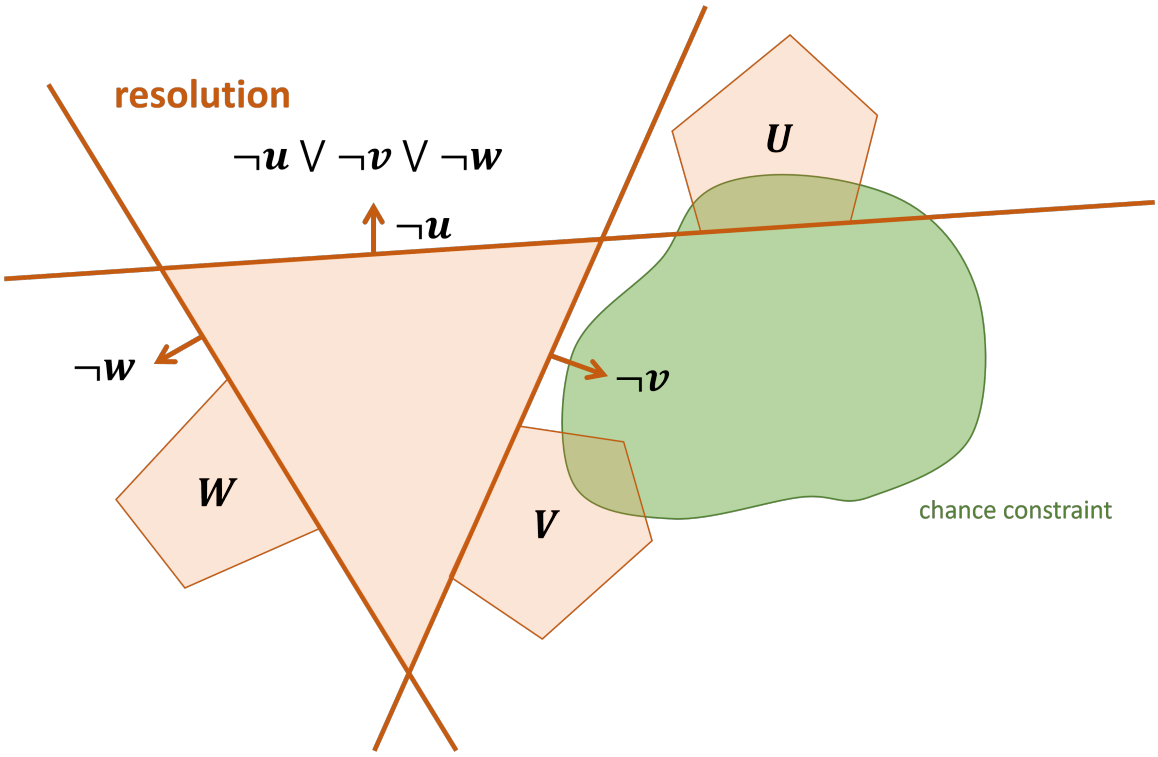

Figure 5-15: The resolution of a Level 2 *infeasible* conflict also creates a convex polytope obstacle in the risk allocation space. By definition, branching on any one of this obstacle's facets would contradict one of the previously selected facets on this branch in the search tree. Therefore, the enumerator should backtrack instead of branching on this new obstacle.

have to branch on those.

Figure 5-15 depicts the new obstacle this forms in the risk allocation space. In a sense, this new obstacle doesn't contain any new facets, but rather is built from the negation of existing ones. The figure makes it clear, though, that if were to continue branching on this new obstacle, we would hit a dead end in every case. Since the current branch already contains linear constraints u, v, and w, then branching on any one of $\neg u$, $\neg v$, or $\neg w$ will result in infeasibility.

Again, this should be detected by any logical inference or propagation procedures in the enumerator. In this chapter, we have treated the enumerator as a black box, just like the NLP solver in Level 2 and the STNU controllability checker in Level 1. However, since the NLP solver in Level 2 is expensive to run, it is worth considering pruning strategies to reduce the number of times Level 2 is called. Section 6.4 addresses how others have handled similar situations.

Chapter 6

Related Approaches

In this chapter, we review how others have provided scheduling solutions for scheduling plans with temporal uncertainty. Primarily, we consider prior efforts to provide scheduling solutions for pSTNs, and we identify their advantages and disadvantages relative to our work. Like ours, many of those research threads also began with static policies, and then turned to dynamic policies, recognizing the value of online flexibility. There is less consensus, though, on how to formulate the probabilistic condition of the problem. Some threads have initially minimized the total risk of violating temporal constraints, and then switched to chance-constrained formulations, while others have transitioned from chance-constrained to risk-minimizing.

We begin by briefly recounting the history of developments in STNU scheduling in Section 6.1. Though the foundational theory of STNU controllability has been documented throughout this thesis, particularly in Sections 2.1, 4.1, and 4.3, it is valuable to situate it within this history. Then in Section 6.2, we review early efforts to schedule pSTNs, which occurred in parallel with those STNU developments.

The thread of work that formally defined chance-constrained scheduling, and which led directly to this thesis, has also been extensively documented as part of this thesis, namely, in Chapters 3 and 4. Therefore, we do not repeat it here. Instead, in Section 6.3, we review some heuristic methods that were published contemporaneously with that work. These methods aim for speed and ease of computation, and produce sound policies, but so far lack theoretical analysis of their associated incompleteness.

Finally in Section 6.4, we relate our work to other efforts in developing conflict-directed hybrid algorithms. The problems they solve aren't necessarily about scheduling, but they share similar mathematical structure as ours, and hence similar solution techniques apply. Sections 4.2 and 5.1 already touched on some of their techniques, but here we give a more comprehensive overview.

6.1 STNU development history

The problem of STNU controllability along with its flavors of strong, dynamic, and weak, was introduced by Vidal and Fargier when they generalized STNs to STNUs [60]. Strong controllability was quickly shown to reduced to STN controllability, and at that time, dynamic controllability was hypothesized to be in PSPACE, due to its "policy versus Nature" aspect. A couple years later, Morris and Muscettola collaborated with Vidal to show that a pseudo-polynomial time algorithm existed [42].

Morris and Muscettola continued working to bring down the polynomial exponent to $O(n^5)$ and then $O(n^4)$ [41] [38]. The $O(n^5)$ result was enabled by defining the distance graph form of STNUs, where lowercase and uppercase edges were introduced. By rewriting the original reduction rules in terms of the distance graph edges, one could prove that only a polynomial number of edge generation rounds were needed, hence leading to the $O(n^5)$ algorithm. The $O(n^4)$ result was based on the discovery that semi-reducible negative cycles (SRNCs) were the key analogue to STNs' negative cycles. Thus, entire extension paths could be traced and reduced against lowercase edges, thereby bypassing multiple rounds of edge generation.

Several years later, Morris further brought that runtime down to $O(n^3)$, acknowledging its basis in insights from Muscettola [39]. The clever idea was to discover extension paths *in reverse* rather than tracing them forward. In a nutshell, extension paths are distinguished by their last edge having negative weight of sufficient magnitude. Thus, we treat every negative edge as a potential termination of a extension path, and since the remaining edges are non-negative, we run Dijkstra backwards on them to find the most negative, or "strongest" possible extension paths. The proof of correctness is more intricate, but the

result is an extremely efficient algorithm whose asymptotic runtime approaches that of strong controllability-checking.

Meanwhile, Hunsberger joined the effort with his keen eye for rigor and accessibility, and identified a flaw in the original definition of dynamic controllability [30]. While fixing it, he offered a more intuitive definition based on his own definition of execution histories. After several years steeped in the literature, Hunsberger gave us fully rigorous proofs on the fundamental link between DC and the existence of SRNCs [33], whereas Morris and Muscettola had omitted many details. He also formalized the notion of actually producing a dynamic policy, that is, what decisions it outputs during execution, and what is the overall execution algorithm. He started with an $O(n^4)$ execution algorithm [31] before bringing it down to $O(n^3)$.

Besides the effort on faster DC checking and execution algorithms, others have provided various extensions and alternative formulations. Wah and Xin were the first to encode the conditions of dynamic controllability into a constraint program [66]. They were motivated by the desire to reduce the flexibility available to the policy without making the STNU uncontrollable. This may seem nonintuitive, but their justification was that maintaining scheduling flexibility can increase the overhead of keeping resources on standby. Thus, the variables they were solving for were the $[l, u]$ bounds on the *controllable* durations and requirement constraints. However, the concept is equally applicable when we switch to variable bounds on the uncontrollable durations for our risk allocation.

Although the distance graph form had been recently published, Wah and Xin's encoding was based on the original reduction rules and the pseudo-polynomial algorithm [42]. Since those rules were less unified, the encoding had many special cases. They also directly encoded any conditional constraints into a nonlinear form, so that an NLP solver could handle them. This process adds an auxiliary variable and several new constraints for each such original constraint. Cui et al. [13] later clarified this encoding by explicitly showing the original disjunctive form of those conditional constraints. They also presented an alternative encoding, where they translated the disjunctive form into a mixed-integer linear program (MILP), requiring different auxiliary variables and additional constraints. However, they showed that using a MILP solver on this encoding was faster than prior NLP approach.

The last set of extensions we discuss is for the problem of incremental DC-checking, where we are given modifications to an existing STNU, and expect to receive an answer faster than checking DC from scratch. We say more about this problem in Chapter 8, where we contemplate how to adapt it to pSTNs for future work. For now, we note that incremental consistency checking existed for STNs [52], and Stedl began the effort to port over the ideas to incremental DC [54].

Namely, when an existing constraint is either tightened or relaxed, certain derived edges may need to be further tightened, or invalidated and regenerated. Stedl addressed the tightening case, and Shah carried forward the work to include handling constraint relaxations [51]. Nilsson followed up by identifying a minor flaw in Stedl and Shah's tightening rules [44]. This jumpstarted independent lines of work by Nilsson and Hunsberger, in which they provided ever faster algorithms for incremental DC, both culminating in $O(n^3)$ time algorithms [32] [45].

6.2 Early pSTN efforts

In contrast to the flowering of work on STNU dynamic controllability, relatively few have addressed the extension to pSTNs. This section and the remaining ones in this chapter document those efforts. Notably, all of them effectively "allocate risk" by looking for $[l, u]$ bounds, i.e., chopping off the tails of distributions. Hence, they are also searching for an STNU policy to apply to their pSTNs, whether they express that explicitly or not. What distinguishes them is their unique strategies for assigning such bounds.

Not long after STNUs were introduced, Tsamardinos explored extending them to have probabilistic durations, and proposed several candidate solution methods, but most were not fully developed. He assumed the explicit objective of minimizing risk, and termed this the probabilistic simple temporal *problem* (pSTP). His first and most complete result was to derive an analytical solution for the least risky static schedule [57]. This solution relied on a restriction of the STNU definition at the time, which was that uncontrollable durations must start on controllable events [60].[1] As a consequence, direct sequences of uncontrollable durations were not allowed. Tsamardinos took advantage of the fact that

controllable events existed interspersed between probabilistic durations, so he didn't have to consider compositions of two or more durations' distributions.

Tsamardinos then turned his attention to dynamic scheduling, but success was more limited. He began by proposing a couple heuristic solutions that manually tweaked candidate STNUs. Then he attempted a mathematical programming approach by encoding the DC reduction rules in the form they were known back then [59]. Unfortunately, he was only able to encode the subset of the rules that resulted in unconditional ordinary edges, and not those that yield uppercase edges, which have a conditional interpretation. This was also before Wah and Xin had published their full DC encoding with conditional constraints. Therefore, Tsamardinos's optimization was necessarily incomplete, which he acknowledged, leaving out portions of the solution space that a full DC encoding or checker might be able to access.

The second early effort at pSTN scheduling was due to Lau et al., who termed their problem *robust controllability* [35]. Like ours, their problem is chance-constrained, in that they are trying to satisfy a risk bound ϵ that limits the likelihood of failure. They also aim to produce dynamic policies.

Their solution method is radically different, however. First, they implicitly assume a uniform risk allocation. Second, they explicitly construct their policy, where the execution time of each controllable event is an affine function of the probabilistic duration outcomes that must precede it. Thus, they are solving for the coefficients of this function.

To do so, they construct a mathematical program that requires the total risk of violating constraints[2] not to exceed ϵ. This is straightforward for them because their policy form gives them direct access to expressions for each event's execution time. Thus, the third difference is that they don't write down a reformulated chance constraint. Instead, their policy form actually allows them to compose distributions, and rather than deal with convolving them, they operate on a pair of "deviation parameters" that can be empirically measured for any distribution. Ultimately, the mathematical program is takes the form of a second-order cone program (SOCP), for which they use an external solver.

[1]This was seen as harmless at the time, because inserting extra controllable events with $[0, 0]$ connections doesn't affect dynamic controllability. However, such $[0, 0]$ "fillers" in between uncontrollable durations pose difficulties for strong controllability. Eventually, this restriction was silently dropped.

[2]Recall that no one but us has made the distinction between requirements and activities. Hence, their definition of failure or success applies to them all, not just the requirements.

The major limitation of this approach is the restriction of the policy form. As our sample policy execution in Example 2.11 showed, the decisions of a dynamic policy have no reason to be linear functions of duration outcomes. Furthermore, there is no preset ordering in the arrival of events in the execution history, so it's premature to declare offline which outcomes should affect which controllable events. Therefore, this approach leaves many dynamic policies inaccessible. Nevertheless, it's an interesting formulation, and the principles underlying this offline analysis could be useful when we extend our problem to encompass multiple chance constraints, discussed in Chapter 8.

The other limitation, which the authors acknowledge, is assuming the uniform risk allocation. They offer a heuristic to rebalance risk across the probabilistic durations. However, it only adjusts the risk for one such duration at a time, and thus isn't a comprehensive solution like ours, where we specify a reformulated chance constraint for the NLP solver to perform "variable" risk allocation in one pass.

6.3 Heuristic methods

While Tsamardinos proposed a couple heuristic strategies for risk allocation to obtain pSTN dynamic policies, those weren't further studied at the time, likely due to their lack of theoretical guarantees. More recently, Boerkoel and his students have contributed their own such heuristics, and provided empirical evaluation of their performance. Instead of directly limiting the probability of scheduling failure, these approaches use the proxy of sequentially adjusting bounds on the probabilistic durations, according to custom heuristics. Therefore, they would be unable to find solutions even when they exist, but their relative simplicity likely gives them faster runtimes.

Lund et al. proposed a method to find pSTN static policies, called SREA, that sidesteps the complexity of nonlinear optimization [37]. SREA performs a binary search on a parameter α that specifies the maximum allowable risk per probabilistic duration. (This binary search is reminiscent of one of Tsamardinos's heuristic solutions.) For each setting of α, they remove $\frac{\alpha}{2}$ probability mass from the tails of each distribution, resulting in initial STNU-style $[l, u]$ bounds. Then they encode strong controllability (SC) into an LP (in the

manner of Fang [20]), and push those $[l, u]$ bounds as far out as possible without violating SC. The binary search thus wraps around this process, searching for the minimal α whose corresponding LP is still feasible.

SREA replaces the complexity of solving for a nonlinear chance constraint across the entire plan with this binary search. Within each iteration, risk allocation is achieved by starting with a uniform allocation, and then using an LP to dole out additional risk. The main weakness is that this LP does not consider the probability mass when doing so. Instead the heuristic only measures how much *temporal* uncertainty is captured. Therefore, this strategy limits the space of risk allocations it can access.

In the same work, Lund et al. followed up SREA with DREA, which was their first attempt at dynamic scheduling for pSTNs. DREA performed dynamic dispatch of a pSTN by running *STN* dispatch algorithms [43] [58] on the STNU implied by the risk allocation bounds. Then, whenever an uncontrollable event arrived and got observed, DREA would simply rerun SREA to obtain new windows. The reasoning for this approach was that no matter whether it arrived within or outside the assumed window, it's known value reduced its uncertainty to zero, and thus could inform whether the intervals on the remaining activities should be widened or tightened.

Unfortunately, this purely reactive strategy falls short of a full-fledged dynamic policy, which prepares for such reactivity in advance. It would be analogous to dynamically dispatching an STNU by treating it like an STN, and repeatedly recomputing the schedule on uncontrollable event arrivals. This is even weaker than the pseudo-controllability concept that Morris et al. used as an intermediate step in their first algorithm for checking STNU DC [42]. Therefore, many dynamic policies likely remain inaccessible to DREA.

DREA also suffers from repeated evaluation of LPs online, which can be expensive with respect to the real-time decision-making required of dynamic execution. Abrahams et al. mitigated this by introducing two thresholds for triggering rescheduling [1]. First, they keep a "counter" of how much risk is accumulated by each passing outcome of a probabilistic duration, and they rerun SREA only when that counter exceeds a threshold. The precise semantics of "accumulated risk" is debatable, but the effect is to recalculate the risk allocation only every once in a while. When they do, SREA's binary search may arrive

on a different α per duration, and the second threshold is to let the new risk allocation take effect only if α changed significantly. Together, these two conditions reduce the amount of proactive rescheduling by DREA. This improves its online runtime performance, but also increases the likelihood of failure, due to the decreased reactivity.

The latest contribution comes from Gao and Popowski, who rely on using STNU dynamic policies like we do, instead of providing their own in the style of DREA. They provide two options for finding their risk allocations [22]. Their first initializes a uniform risk allocation, just like SREA, and tries to discover an SRNC in the resulting STNU. Then it tightens any $[l, u]$ bounds in that SRNC in order to resolve it. The tightening method they use is guaranteed to minimize the total *volume* lost in the duration space, but not *probability mass* [2]. It is implied that they repeat this process of finding and tightening SRNCs until the STNU is dynamically controllable.

The incompleteness of this method stems from the fact that they only tighten certain durations' $[l, u]$ intervals without widening others. This monotonically decreases the amount of temporal uncertainty in the STNU, so it is likely that eventually the STNU will become controllable. Thus, there is no real effort to respect or even evaluate a risk bound. To do so, or even to minimize the risk, they would need to wrap their procedure around a binary search on the risk bound, like SREA does. Even then, since there is no guarantee on the order in which SRNCs are discovered, and hence the order in which edges are tightened, there are likely some risk allocations that remain inaccessible.

Gao and Popowski's second option builds on the same concept of discovering and resolving SRNCs, but tries to follow an order, starting with the "riskiest" SRNCs. Again, they rely on uniform risk allocations to begin with, but their insight is that by perfoming binary search on a risk bound α (in the style of SREA), they can identify an α^* below which the STNU first becomes non-DC. Whatever SRNC comes out of the STNU with $\alpha^* - \epsilon$ risk per duration is proclaimed "riskiest", and gets resolved by tightening its durations' intervals to each have $\alpha^* + \epsilon$ level of risk. Those intervals are then frozen, so that the next round of binary search and tightening for the "second riskiest" SRNC won't disturb them.

In summary, both their methods can be viewed as sharing the same high-level strategy as our conflict-directed algorithm, in that they also discover SRNC conflicts and resolve them

sequentially. The difference is they propose custom heuristics to achieve the resolutions, which choose certain $[l, u]$ bounds to tighten. Since this progressively increases the amount of risk taken, they rely on a binary search wrapper to periodically reset the risk to acceptable levels, and there are no completeness guarantees associated with this. Also, their tightening strategies do not explicitly acknowledge the presence of extension subpaths, which could be made non-negative to resolve the SRNC.

In contrast, we derive constraints that fully express the conditions for resolving the SRNC conflicts. We rely on an external NLP solver, plus our own exhaustive combinatorial search, to satify those constraints in our risk allocations. In particular, the solver allows us to simultaneously *reallocate* risk from some durations to others in one step, without falling back on a sequential binary search, which is likely to be incomplete. We thus solve the risk allocation problem exactly, modulo any incompleteness from the solver, which we've shown in Section 4.4 can be reasonably mitigated.

6.4 Conflict-directed hybrid approaches

Central to our approach in Algorithm 4.1 is the usage of an STNU controllability subsolver to discover conflicts that prune away regions of the risk allocation space. Furthermore, the risk allocation obstacles for dynamic controllability conflicts turn out to be convex polytopes. Thus, we resolve conflicts both at a continuous level, using an NLP solver to handle linear resolutions plus the nonlinear chance constraint, and at a discrete level, when choosing which linear facet of each polytope to respect.

This principle of isolating a subproblem that can be efficiently solved, and returning conflicts from it to inform a master problem or master search, has shown up in several other works. It is especially favored when addressing scenarios that involve both discrete and continuous features. Thus, there are two senses of the word "hybrid" when describing such approaches. One is that expressing the problem requires both types of features. The other sense, and the one we focus on, is that the algorithmic solution consists of two or more solvers interacting to progressively cut down on the remaining solution space.

In this section, we identify such works and relate them to our approach. In most of

them, they are solving problems more general than scheduling, and thus employ additional techniques for handling the discrete conditions. Some but not all of those techniques could be applied to our algorithm, and I note when that would be possible.

The common mathematical structure between our work and these others is the disjunctive linear program (DLP), consisting of disjunctions where each clause is a linear constraint. For us, the DLP arises from the alternative resolution options for SRNCs. That is, not only can we make the cycle nonnegative, we can also make certain extension subpaths in it nonnegative, and all of those options are linear.

DLPs were studied by Balas [3] as a more direct way of expressing discrete optimization problems in certain scenarios. Previously, discrete options would have been represented using binary or integer variables, which can significantly increase the dimensionality of the problem. By working with the disjunctive form directly, Balas derived various LP relaxations along the branches of the search tree, so that a standard branch-and-bound strategy could avoid expanding to the entire depth of a tree. His relaxations typically have interpretations as convex hulls of subsets of the convex polytope obstacles.

Krishnan [34] and Li [36] merged the techniques of conflict-directed forward search for optimal constraint satisfaction [72] into the branch-and-bound framework for DLPs. Namely, they detect conflicting subsets of linear constraints within the LP relaxations along the branches and at the leaves of the search tree. In turn, these conflicts' resolutions yield additional disjunctive linear constraints, which provide options for further tightening the LP relaxations on other branches of the tree. By applying these tightenings before proceeding with branching on the remaining disjunctions of the original problem, this can significantly reduce the depth of forward search.

While Krishnan omitted details on generating conflict sets, Li demonstrated that they could be efficiently extracted when solving the dual LP. This is essentially how Benders' cuts are extracted [4], which were discussed in Subsection 4.3.1 as an interpretation of our strong controllability conflicts. Thus, our static algorithm shares the same underlying theory of conflict extraction. The difference is we use an alternative algorithm that is tailored for STNU distance graphs, and therefore has a natural counterpart in dynamic scheduling, which is less easily expressed in terms of Benders' decomposition.

One aspect of DLPs not present in our problem is having to optimize an objective function. Hence, we wouldn't gain any pruning by applying the branch-and-bound framework, which is meant to eliminate suboptimal subtrees. However, Chapter 8 discusses the possibility of applying optimization to our problem, i.e., finding optimal scheduling policies. In that case, the techniques of Balas and others could be helpful. For instance, Li also generalized infeasibility conflicts for DLPs to *suboptimality* conflicts. That is, when a relaxed LP at a node is feasible but whose solution is worse than the incumbent's, we can identify *which clauses* led to the solution being suboptimal. So, not only do we prune the current subtree, as branch-and-bound would, we also prune potentially similar subtrees on other branches without having to discover them.

Another difference between the DLP literature and the DLP in our problem is that we are not given the DLP upfront, but rather we build it from the disjunctive conflict resolutions we learn along the way. That is, we start with *no* constraints in our Level 3, and only when an appropriate SRNC is discovered in Level 1, do we generate the disjunctive resolution in Level 2 and pass it up to Level 3. Thus, our search tree in Level 3 is analogous to the branching on conflicts that Krishnan and Li perform. We do lack the concept of solving relaxed LPs at each node in our search tree, though it seems this could be easily incorporated.

Ultimately, DLP forms only a subset of our problem, addressed by Level 3. We also have a nonlinear chance constraint and an STNU for which to check controllability, which are handled by Levels 2 and 1, respectively. Relatively recently, Ono [46], Yu [74], and Fang [19] have solved problems with similar features, so we conclude this section by relating their work to ours.

Ono's problem was that of chance-constrained path planning, where the goal is to avoid obstacles with a bounded probability of collision [47] [46] [48]. This was the context in which risk allocation was first developed for addressing chance constraints. In path planning, since collision risk arises from uncertainty in vehicle dynamics, risk is allocated to *random state variables* indicating the vehicle's position along its nominal path. In our scheduling problem, there is no single "trajectory", so the "random variables" to which risk is allocated are the probabilistic durations.

Ono modeled obstacles as convex polytopes, thus requiring a DLP to avoid those regions.

(Any nonconvex obstacle can be modeled as a union of convex polytopes.) Like our problem, he also had a nonlinear chance constraint, so at the leaves of the search tree, an NLP needs to be solved. He called solving this *convex risk allocation*, since the chance constraint turns out to be convex for reasonable risk bounds and uncertainty distributions [8].

Ono also solved relaxed problems on the non-leaf nodes, but here he avoided the expense of NLP by applying a *linear* relaxation to the chance constraint. If the objective function is linear as well, or even piecewise linear, then he only needs to solve inexpensive LPs to prune suboptimal subtrees. Thus, Ono's method mirrors prior branch-and-bound approaches to solving DLPs, with the exception of an NLP at each leaf. He does not, however, employ the conflict-directed methods of Krishnan or Li.

Yu addressed the problem of relaxing temporal plans, where to restore the schedulability of a plan, certain requirements need to be relaxed or dropped [77] [75] [74]. Due to the shared context of scheduling, this work has many parallels to ours. First of all, the problem of relaxation is inherently about resolving existing conflicts. Thus, Yu discovers temporal conflicts of the same form we do, that is, negative cycles in the distance graph. It follows that his dynamic scheduling conflicts are also SRNCs which require disjunctive linear constraints to resolve.

Yu's temporal plans have built-in alternative subplans as relaxation options, and these alternatives are associated with user preferences. To process conflicts and optimally restore feasibility, then, he leverages the conflict-directed A* algorithm of Williams and Ragno [72], just as Krishnan and Li did. He augments the algorithm by allowing scheduling conflicts to be resolved not just by switching to different subplans, but also by adjusting bounds on the requirements and activities' durations. These bounds are directly analogous to the risk allocation variables in our problem, and thus his method of finding optimal relaxations is structurally similar to our algorithm. If the user preferences on those bounds are linear, then an optimal relaxation can be found by solving an LP, followed by checking for temporal controllability. Later, Yu incorporated a relaxable chance constraint, which turns the LP into an NLP [76].

Finally, we consider Fang's work [19], which can be viewed as a generalization of Ono's. While Ono was solving a chance-constrained DLP, Fang swaps the DLP for the more

expressive mixed-logic linear program (MLLP), where the continuous linear constraints are conditioned on boolean variables or logical formulas. Thus, the search tree branches on those boolean variables, which activate various linear constraints along the way. At the leaves, the original cc-MLLP becomes a chance-constrained linear program (cc-LP).

Like Ono, Fang handles the chance constraint by solving relaxed LPs instead of exact NLPs along the branches. Notably, though, Fang does not solve an NLP even at the leaves. Instead, he simply solves for lower and upper bounds on the random variables, and then evaluates whether the "risk allocation" they form respects the chance constraint's risk bound. This is also an LP, and if its solution violates the chance constraint, then a cutting plane conflict is generated based on how much we went over the allowed risk bound.

This strategy of generating conflicts from inexpensive evaluations of the chance constraint is a unique and elegant alternative to directly solving the chance constraint. It is a different application of the same spirit in our strategy, where rather than fully encoding STNU controllability, we simply check it and extract conflicts. In Fang's case, when the LP solution at the leaves violates the chance constraint, a linear conflict resolution is derived. But if the LP or the relaxed LPs on the branches are infeasible, then an irreducible infeasible subset (IIS) is obtained, in a similar manner as Li's conflict extraction. These form additional disjunctive linear resolutions, just like how our SRNC Level 1 conflicts and risk allocation Level 2 conflicts are handled by Level 3.

In summary, our work incorporates elements of Ono's, Yu's, and Fang's solutions for their respective problems. Namely, we share with Ono the need to respect a nonlinear chance constraint while avoiding convex polytope obstacles, we share with Yu the form of scheduling conflicts and resolutions, and our solutions for static and dynamic scheduling parallel Fang's solutions for solving cc-LPs and cc-MLLPs. Some aspects of their algorithms could also be incorporated into ours, such as solving relaxed problems on the branches, and finding optimal solutions, which we leave for future work.

Chapter 7

Empirical Validation

The major claim of this thesis has been that chance-constrained dynamic policies are key to enabling the execution of large, complex plans. After formally defining this problem in Chapter 2 and reformulating it in Chapter 3, we built up an algorithmic solution to it in Chapters 4 and 5. Thus, we now have the ability to provide policies of the desired form. The remaining questions to evaluate the claim fall into two categories: a) how well do our policies scale with plan size, and b) how efficiently can we generate our policies. The first category is about how the increased richness of our problem captures more solution space, while the second is about the runtime performance of our algorithm. The two sections of this chapter present empirical evidence that support the overall claim with respect to these two categories.

Section 7.1 addresses the solution space question by comparing how many of the test scenarios for which we can find static versus dynamic policies. We find that the success rate of finding static policies quickly falls with problem size, while we can continue finding dynamic policies at the same rate for problems at least 10 times larger. We also evaluate the improvement when allowing flexibility in allocating risk over fixing it to a uniform allocation. These gains are less dramatic, but for dynamic policies, we can still solve about an extra 10% of the scenarios. We perform these comparisons on a lunar construction scenario inspired by extra-vehicular activities conducted by NASA astronauts.

Section 7.2 then addresses the efficiency of our algorithm in coming up with such policies. This pertains to the conflict-directed strategy we adopted in Chapters 4 and 5 as an

alternative to employing the full encoding of STNU controllability discussed in Section 4.1. The end goal of both is to produce chance-constrained policies, so we will not be comparing against STNU algorithms, which are much faster due to not having to solve a nonlinear continuous constraint.

Based on Subsections 4.1.1 and 4.1.2, it is reasonable to find a solver that can handle an encoding of strong controllability plus the chance constraint, but much harder when we swap out the strong controllability conditions for dynamic controllability. Thus, we descoped implementing the full encoding approach to generate chance-constrained dynamic policies for our experiments. However, we observe that our conflict-directed algorithm in Chapter 5 only slightly increases in runtime when switching from static to dynamic policies. Meanwhile, the full encoding for static policies is typically at least an order of magnitude (10 times) slower than the conflict-directed version. This shows that our conflict-directed algorithm for dynamic policies would be at least similarly competitive against a full encoding.

We review the results previously presented in [69] comparing the full encoding and conflict-directed runtimes for static policies, but now explain them in more detail. The benchmark scenarios are to manage a fleet of vehicles for a car-sharing company.

I begin both sections by describing the experiment setup. This includes the form of algorithm implementation and the benchmark scenarios that were generated For the benchmarks, I identify the various parameters that impact their problem size and complexity. Then I present graphs of certain metrics as those parameters vary, increasing the problem size, and I draw conclusions from the trends displayed.

7.1 Solution space evaluation

The ideal metric for evaluating the expressivity of chance-constrained dynamic policies would be how much of the space of scheduling policies we can access. This is depicted in Figure 7-1, which is based on Figures 3-11 and 3-13 from Subsection 3.3.1. Unfortunately, we already showed in Chapter 3 that the policy space is intractable to work with, let alone measure. Instead, we choose to measure how likely it is that our algorithm can find such

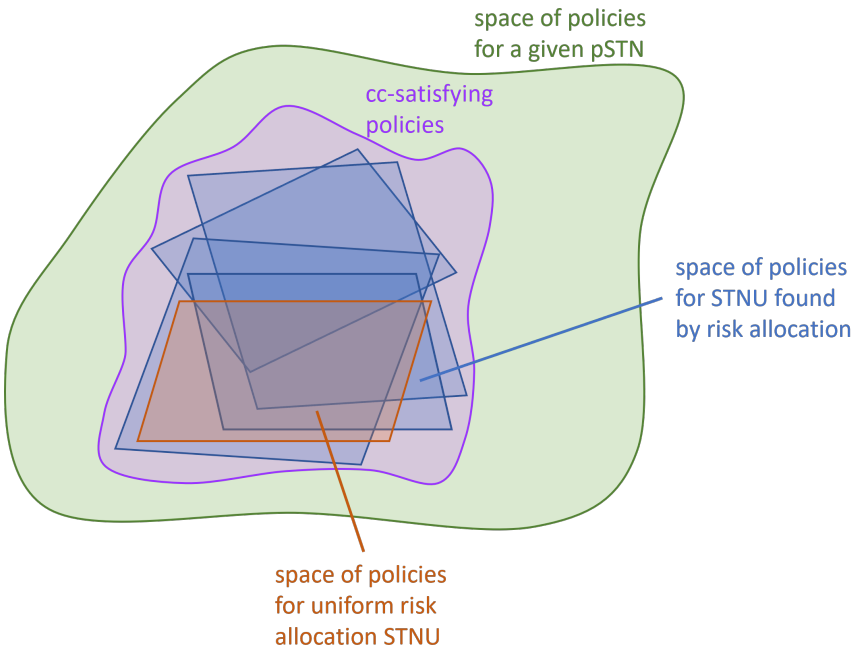

Figure 7-1: We are interested in how much of the entire solution space of chance-constrained policies (in purple) we can access. While our problem reformulation prevents us from finding every policy, its flexible risk allocation approach lets us access the policy spaces of various proxy STNUs (in blue). In contrast, a fixed uniform risk allocation approach is limited to only one such policy space (in orange). Thus, the ideal metric would be how much of the purple space is covered by the union of the blue regions, compared to just the orange region.

a policy solution, across a set of randomized scenarios. To see why this is a reasonable alternative, we consider what the ideal metric in the policy space really means.

First, note that in the absence of a flexible strategy for allocating risk where it is needed most, a reasonable way to address a chance constraint would be to use a uniform risk allocation. That is, we split the chance constraint's Δ risk bound equally among all the probabilistic durations in a pSTN, and cut off tails of equal probability mass for both the lower and upper bounds. This results in what we call the *uniform risk allocation STNU*, and there is no guarantee that it is controllable. Since this approach sidesteps all the risk allocation machinery of our cc-pSTN algorithms, for brevity we call this the "STNU" approach when presenting our experiments, even though it still technically addresses a cc-pSTN problem.

In Figure 7-1, recall that the purple region represents those pSTN policies that satisfy

the chance constraint. For the uniform risk allocation STNU, its space of feasible[1] policies would be a subset of the purple region, represented as an orange parallellogram.[2] In contrast, our risk allocation approach to the cc-pSTN allows us to consider many different proxy STNUs for the pSTN, all satisfying the reformulated chance constraint. The feasible policies for these STNUs thus all fall inside the purple region, within their own blue parallelograms. In actuality, there are an infinite number of such parallelograms, and the space of chance constraint-satisfying policies that we can access is the union of them all. This union includes the orange parallelogram, because that STNU respects the reformulated chance constraint as well.

This relationship between the orange, blue, and purple regions holds regardless of whether we are looking for static and dynamic policies. Hence, our figure is applicable in both cases. It is also important to note that the solution spaces for static policies are subsets of those for dynamic policies. This is because static policies, as shown in Definition 2.14, can be considered special cases of dynamic policies, so every strongly controllable STNU is also dynamically controllable.

Something that Figure 7-1 assumes, though, is that such orange and blue regions actually exist. That is, it is not guaranteed that our risk allocation approach will find *any* proxy STNUs that are controllable. Likewise, the uniform risk allocation STNU has no guarantee of being controllable, and if not, then no orange region exists either. For both the blue and the orange regions, it is also the case that there might exist dynamically controllable STNUs, but none that are strongly controllable. Thus, as we make the pSTN more constrained by tightening its requirements or widening the uncertainty on its probabilistic durations, the number of parallelograms in Figure 7-1 will drop until there are none at all.

While we don't have tools to directly measure how much of the pSTN's policy space is covered by the orange or blue regions, our algorithm does report whether we are able to find[3] such regions. Additionally, because we know that any orange region must be a subset of the union of the blue regions, this means as we tighten the pSTN, the orange region will

[1]The STNU policy space, in all its richness, is not actually bounded by a parallelogram. This is just meant to remind us that STNU policies can rely on the bounds of their uncontrollable durations, whereas pSTN policies have to allow for the possibility of them going to $+\infty$.

[2]To reduce clutter, I only display the feasible policy regions for STNUs.

disappear before the blue union does. To be sure, the purple region of all chance-constrained solutions also shrinks as the pSTN tightens. However, it will disappear last after the orange and blue regions have been reduced to the null set, even though we will not be able to detect exactly when.

If we perform a "sweep" over these pSTN tightenings, then we can observe how much earlier STNU algorithms fail to find a solution compared to our algorithm for solving the cc-pSTN model. We can also perform the same comparison for static versus dynamic policies. This can be viewed as an indirect measurement of the solution space coverage. Unlike the uniform risk allocation STNU, our risk allocation approach isn't limited to that one parallelogram. Rather, it can find other STNU policy spaces that fit in the shrinking purple region, and that wiggle room accounts for its additional coverage of the solution space.

7.1.1 Experiment setup

For our benchmark scenarios, we model the task of erecting satellite communication dishes on the moon These are adapted from the benchmarking scenarios of Bhargava and Pittman [6] [49], which in turn were inspired by operational knowledge of spacewalks, combined with NASA's Artemis program for lunar exploration. The goals is to assemble an array of satellite dishes, and a team of astronauts can parallelize the activities. However, there are also some coordination requirements which could introduce extra waiting times, and thus endanger the overall deadline requirement.

The plan structure for these scenarios is shown in Figure 7-2. Each satellite dish's assembly task is encapsulated in a blue unit consisting of three activities in sequence. First, the astronaut drives to the installation site in a rover vehicle $A \dashrightarrow B$. Then, they perform the actual installation $B \rightarrow C$. Next, they confirm that the dish is working by verifying communication with mission control on earth $C \dashrightarrow D$. When that's completed, the astronaut performs any final wrap-up for the installation $D \rightarrow E$.

The installation and wrap-up tasks, where the astronaut is working solo, have controllable

[3]Due to the incompleteness of the NLP solver in practice, this is distinct from whether such regions *exist*. However, since we use the solver as a black box, this is the best we can do to approximate the criterion.

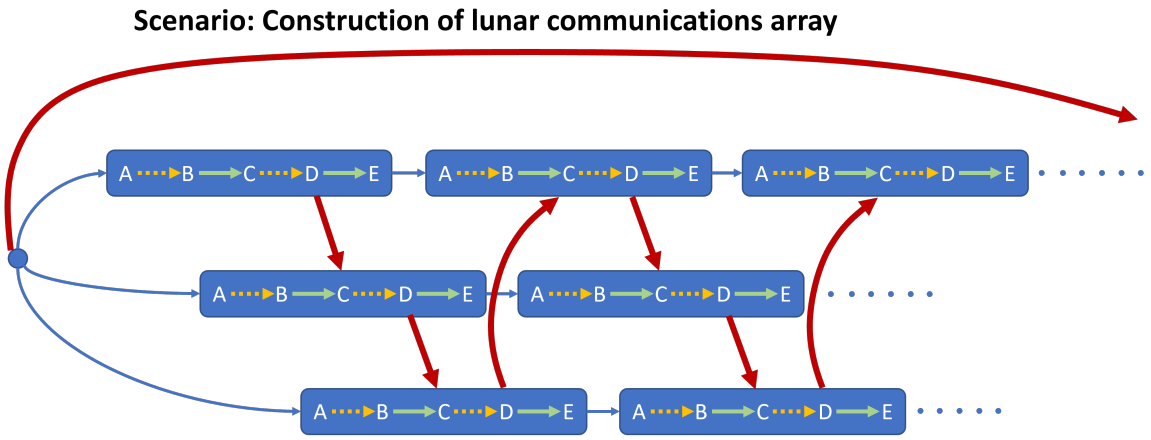

Figure 7-2: Each satellite dish's assembly task includes three activities, the outer two of which have uncontrollable or probabilistic durations. An astronaut is assigned a series of dish assemblies, forming a single "thread" of execution. The inter-thread constraints between astronauts is that the final activities of each assembly need to be performed in series, resulting in a staggered flow of assemblies between the astronauts. There is an overall deadline constraint for completing all assemblies.

durations, each with their respective $[l, u]$ bounds. Meanwhile, the driving time is subject to disturbances from the terrain, and the confirmation step could be delayed by electromagnetic noise and network congestion. Those activities' durations are modeled probabilistically as normal distributions parameterized by respective means μ and standard deviations σ. Technically, such distributions have probability mass below $t = 0$, but as long as μ is at least four or five σ above 0, then that probability mass is vanishingly small with respect to a chance constraint risk bound on the order of 10%.

To reduce the total time required to assemble all dishes, a team of n astronauts is sent, each independently assembling their own m assigned dishes. Each astronaut's timeline is represented as a thread of these assembly tasks linked by $[0, +\infty)$ wait activities. Additionally, a mission requirement imposes some degree of coordination between their threads. Namely, the centralized mission control center wishes to verify the dishes in a rotating sequence among the n astronauts. Therefore, from mission control's perspective, they need to see all the $C \dashrightarrow D$ activities happen in a precise order. This is achieved by introducing $[0, +\infty)$ requirements from the end of each confirmation to the start of the next. Finally, the entire plan has an overall deadline of Tm minutes, where T represents the nominal time it takes to install a single satellite dish, plus some slack.

In summary, the plan structure and complexity is parameterized by n astronauts, m installations per astronaut, and T for the nominal time allotted per installation. If we keep the chance constraint's risk bound constant while increasing n and m, then the amount of risk allocated to each probabilistic duration's tails decreases. This results in wider STNU intervals, both for the uniform risk allocation STNU as well as any others that our NLP solver finds. As a result, those scenarios will be less likely to admit a policy solution. Increasing T, on the other hand, gives the installation tasks more slack in case an activity takes unusually long. That should result in the opposite effect of postponing the dropoff, in percentage of scenarios solved, until larger problems.

For any given problem size and deadline, we are interested in how likely we can find a scheduling policy that respects the chance constraint. However, for any particular $[l, u]$ bounds and (μ, σ) parameters on the activities' durations, we get a binary yes or no. Therefore, we randomly generate those parameters, once per trial, and look at what percentage of trials succeeded in finding a policy.

For the installation activities, the lower bounds are always 0 and the upper bounds are selected at uniform from the interval $[5, 10]$. For the wrap-up activites, their lower bounds are uniformly selected from within $[0, 5]$. Their upper bounds take the form $u = l + \overline{u}$, where $\overline{u}$ is uniformly selected from within $[12, 22]$.

The normal distribution parameters (μ, σ) of the driving and confirmation activities are based on slight perterbations to nominally specified parameters $(\overline{\mu}, \overline{\sigma})$. First, we select σ uniformly from the range $(1 \pm 0.1)\overline{\sigma}$. Then we select μ uniformly from $\overline{\mu} + (1 \pm 0.1)\sigma$. For driving, we choose $(\overline{\mu}, \overline{\sigma}) = (10, 2)$, and for confirming, we choose $(\overline{\mu}, \overline{\sigma}) = (8, 2)$.

For each benchmark trial, four different algorithm implementations were used to solve it. First, we solve for static policies, using Vidal's strong controllability algorithm [61] [60] for the uniform risk allocation STNU, and our two-level algorithm from Chapter 5 for the cc-pSTN. Then, we do the same for dynamic policies, using Morris's $O(n^3)$[4] algorithm [39] and our three-level algorithm. All the algorithms are implemented in Common Lisp, except for the external NLP solver, for which we use Ipopt [64] via a custom foreign function interface. The experiments were carried out on a third-generation desktop Core i7.

[4]Recall that this n refers to the number of events in our temporal network, and not the number of astronauts

7.1.2 Results

To cover a wide range of problem sizes and complexities, we vary the number of tasks per astronaut m from 1 to 50, and the number of astronauts from 2 to 5. Nominally, we set $T = 50$. However, as the number of astronauts increases, the confirmation ordering requirements impose more coordination, which effectively increases the T necessary to keep the plan controllable. For example, if astronaut 1 is having difficulty completing the confirmation for a particular dish, then astronaut 2 will have to sit idle, effectively increasing the total time for its installation task.

In our experiments, keeping T constant at 50 resulted in precipitous dropoffs in the percentage of scenarios solved while the problem size was still relatively small. Therefore, to be able to observe trends over the entire range of m, we bumped up T for the larger n. Namely, we use $T = 65$ for $n = 4$, and $T = 80$ for $n = 5$. In the real world, this would correspond to mission control increasing the built-in slack to accomodate the increased complexity of coordinating more astronauts in parallel.

We ran 100 trials for each combination of n, m, and T, and recorded the percentage of scenarios solved in Figures 7-3 through 7-6. The static policy results for the uniform risk allocation STNU and the full cc-pSTN approach are plotted in yellow and orange, respectively. For dynamic policies, the analogous results are given in gray and blue, respectively. Across all four graphs, the far right corresponds to the maximum number of installation tasks, ranging from 100 to 250. Furthermore, there are two uncontrollable activities per task and four activities total. This means we are solving problems containing up to 500 probabilistic durations, and 1000 activities total.

The most significant trend is that dynamic policies far outperform static policies in terms of scenarios solved. Even the uniform risk allocation STNU has dynamic policies for significantly more instances than our full algorithm could find static policies for. This dramatically highlights the value of having online flexibility to open up the solution space, and thus underscores the importance of extending previous work in chance-constrained static scheduling [20] [69] to dynamic policies. Due to the vast differences in performance between finding static versus dynamic policies, I consider them separately.

in our scenario.

Across all four plots, we are initially able to find static policies for a high percentage of scenarios, but the success rate quickly drops off as we approach 5 or 10 tasks per astronaut. With two astronauts, the cc-pSTN solution space vanishes just after 10 tasks, and it steadily shrinks to 5 tasks as we increase the number of astronauts to five. Observing horizontally, we see that our cc-pSTN algorithm can generally solve the same portion of scenarios at twice the number of activities as the strong controllability algorithm for the fixed STNU. Given the sharp slope in the dropoff, though, this means when observing vertically, for the same problem size, our flexible risk allocation approach solves many more times instances than the fixed uniform approach does.

It is also notable that at small problem sizes, the fixed STNU approach finds static policies for much fewer problem instances than full cc-pSTN approach does, which still has high success rates. However, this observation is tempered by noting the following artifact. At low problem sizes, e.g., less than 5 tasks per astronaut, the trends are not very smooth and can vary wildly. This is true for both the static and dynamic cases, where the number of successful instances is sometimes puzzlingly low at first before shooting up dramatically. Most likely, this is because when there are only one or two tasks per astronaut, an outlier in the generated parameters for the durations can derail us from meeting the overall deadline, which is a low multiple of T. However, as the number of tasks increases, those outliers get smoothed out, via the Central Limit Theorem, and T becomes a better representation of the average duration per task.

For dynamic policies, the trends are dramatically different. First, we see that the algorithms are completely successful for the entire range of problem sizes when there are only two astronauts. Only when we bump up to three astronauts do we start to encounter scenarios for which we can't find a policy, and mostly when using the fixed STNU approach. The full cc-pSTN algorithm doesn't reach that threshold until just below 45 tasks per astronaut, which corresponds to about 130 installation tasks total. By the maximum problem size of 50 tasks, the fixed STNU performance has dropped to about 80%, while the cc-pSTN is at 90%. Thus, the dropoffs are much shallower than with the static policies.

When we move on to four and five astronauts, the dropoffs get gradually sharper, just as they did for static policies. Partly this is due to the total number of activities increasing at

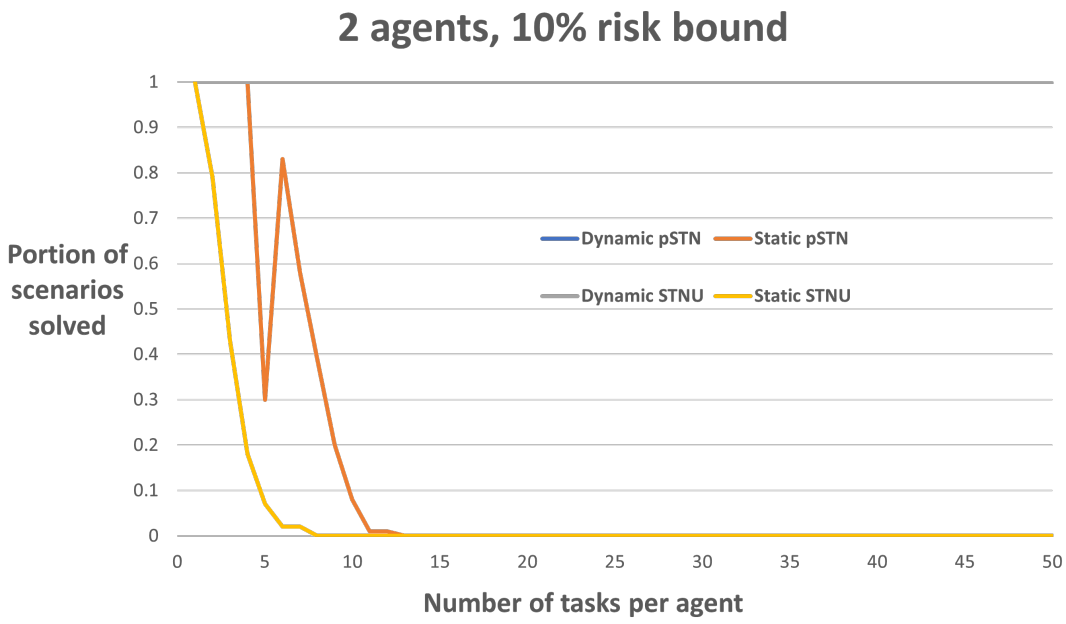

Figure 7-3: When the plans involve two astronauts, all scenarios admit dynamic policies, chanced-constrained or not, while the solution space for static policies falls off around 5 and 10 installation tasks per astronaut, for the uniform risk allocation STNU and the full cc-pSTN, respectively.

a faster rate for every additional task we assign to each astronaut. The other contribution is from the increased number of cross-thread coordination requirements. This is evident from the slopes for five astronauts being more than 25% steeper than the slopes for four. In the five-astronaut case, the success rate eventually plummets to near 0 by the time we reach 50 tasks, whereas for four astronauts, they still hover near 50%.

The shallower nature of the degradation for dynamic policies is further proof of the benefits of online dispatching flexibility. In fact, it is surprising to see the the fixed STNU policies perform so well that the full cc-pSTN policies don't offer as much improvement as they did in the static case. Still, the cc-pSTN policies typically solve 10% more of the total scenarios than the STNU policies do, which is not insignificant. Also, due to the shallower slopes, when observing horizontally at the same level of success, we can solve problems with a larger number of tasks than in the static case.

Although the next section formally addresses runtime, it is with respect to a different set of benchmark scenarios, so I close this section with a brief note about the runtimes of these

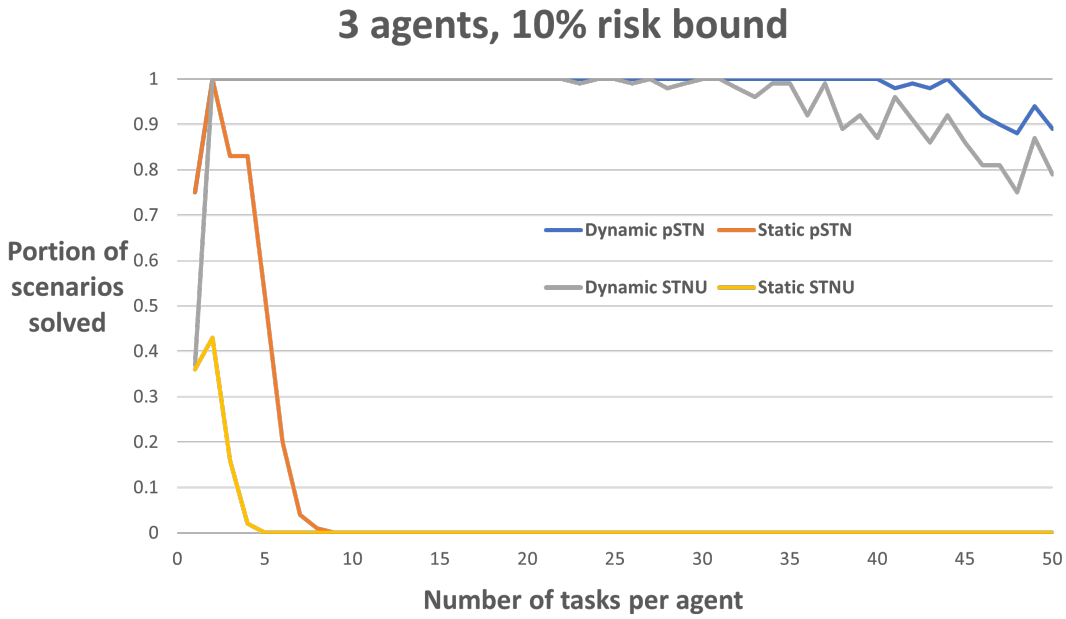

Figure 7-4: When the plans involve three astronauts, some fixed STNUs begin to be not dynamically controllable around 30–35 tasks per astronaut, while the dropoff for flexible risk allocations doesn't begin until after 40 tasks. Compared to having two astronauts, the static policy spaces vanish slightly earlier, a trend that continues for higher numbers of astronauts.

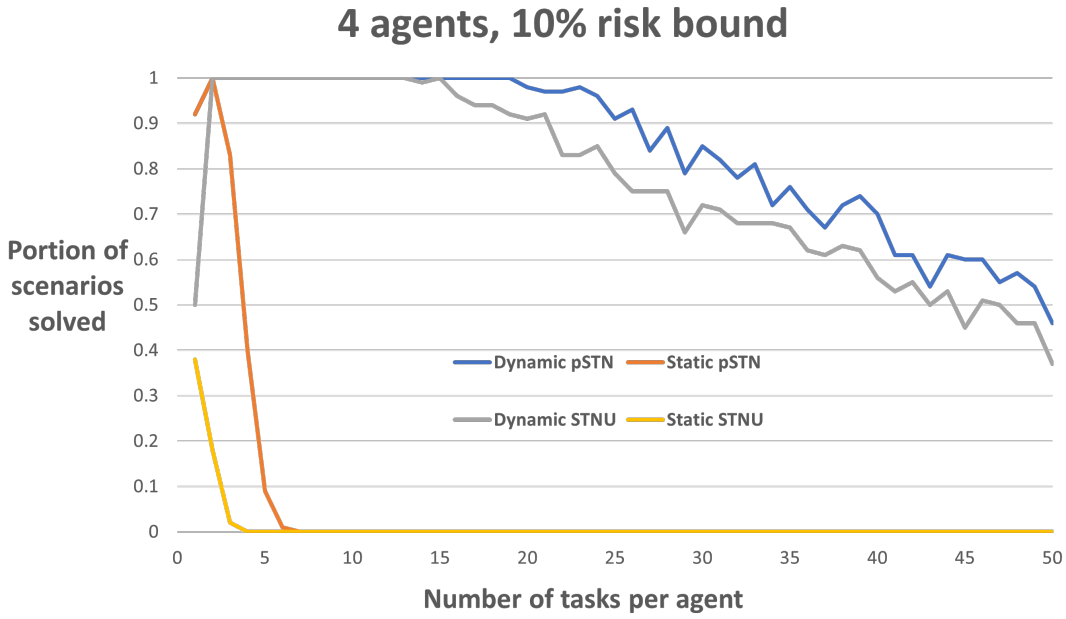

Figure 7-5: For four astronauts, the dropoff for dynamic policies begins around 15 and 20 tasks per astronaut, and we are able to find cc-pSTN policies for about 10% more of the total scenarios.

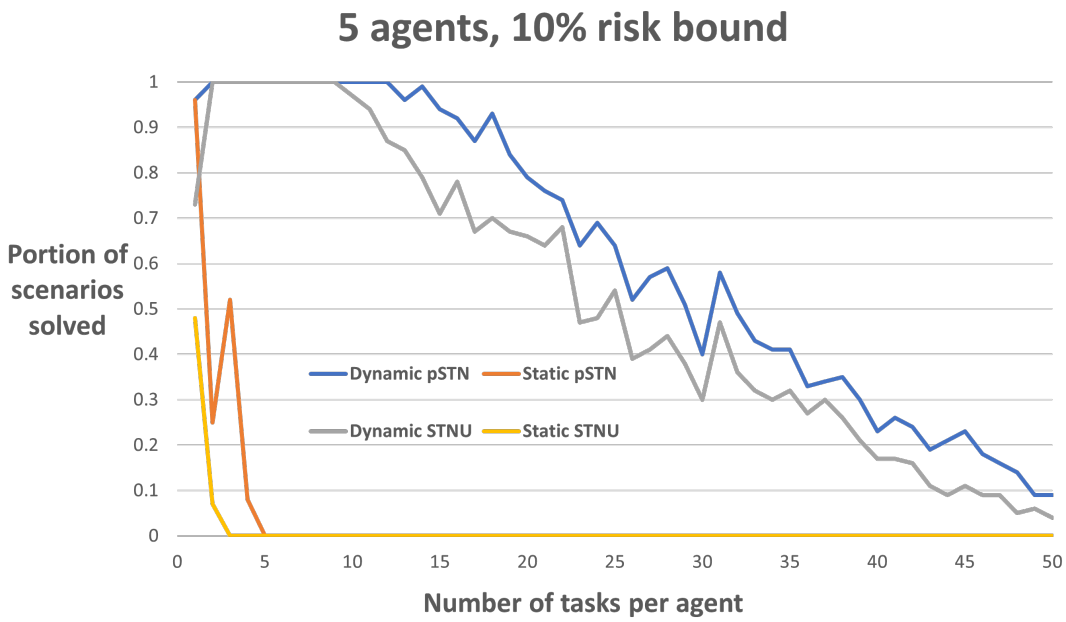

Figure 7-6: At five astronauts, the trends are similar but sharper, and by 50 tasks per astronaut, the dynamic policy spaces are nearly gone.

benchmarks. For our cc-pSTN algorithm, the vast majority of the computational time was spent within the NLP solver. Therefore, the key factor that affects total runtime is how many times the NLP solver is called. For static policies, this corresponds exactly to the number of negative cycle conflicts discovered. For dynamic policies, it is at least that number, but may include more due to combinations of alternative SRNC resolutions.

In our observations for the smaller problems where $m \leq 20$, all cases in which a policy was found terminated after fewer than 10 calls to the solver, and usually no more than 5. These low numbers are a good sign that the discovered conflicts were very informative in blocking out infeasible portions of the chance constraint region. When a policy was ultimately not found, though, our algorithm might discover up to 20 or 30 conflicts before returning false. This suggests that the NLP solver was getting stuck in local gradient fields, and while there may have been an actual solution, it was unable to find it.

For the largest problems, where the number of risk allocation variables approach 1000, each call to the solver could take up to two minutes to complete. Therefore, to avoid needless spinning, we cut off the algorithm after 10 discovered conflicts, and assumed the answer was false. It is possible that we thus missed out on solutions that we could have found if

we had kept discovering conflicts, but the difference is likely to be small. The effect on the graphs would have been to push the orange and the blue lines slightly to the right.

In summary, the results in this section suggest potentially two different strategies depending on whether we are searching for static or dynamic chance-constrained policies. For static policies, using a fully flexible risk allocation approach is a clear win over the uniform risk allocation. Even though we take on the extra cost of running an NLP solver, if we're already solving for a static policy offline, then presumably we can afford that cost. If we are looking for dynamic policies, then in most cases, the policy is still being computed offline, and the extra 10% likelihood of finding a solution is worth the full search for a variable risk allocation. However, if we are in a real-time situation, or if computational power is limited, then applying a uniform risk allocation is a reasonable strategy for producing an answer quickly.

7.2 Runtime evaluation

When evaluating runtime, the claim in question is that our conflict-directed strategy of discovering negative cycles to resolve is more efficient than employing a full encoding of STNU controllability. To perform this comparison, we would need a solver that can handle the full encoding in conjunction with our nonlinear reformulated chance constraint. As we have shown in Chapter 4 previously, an NLP solver suffices for obtaining static policies, but we would need a MINLP-style solver for dynamic policies. The DC encoding would be complex to specify, and we can also expect general-purpose MINLP solvers to have significantly longer runtimes than even NLP solvers. Therefore, we limit our comparison to only static policies.

From our experience running the satellite installation scenarios in Section 7.1, the observation that the solver was called fewer than 10 times per trial held true for both static and dynamic policies. This suggests that our three-level algorithm for producing dynamic policies is not much more expensive than our two-level static version. Combined with the expectation of long MINLP-solving runtimes, any difference in our static comparsion would only be magnified if we had been able to compare in the dynamic case as well.

7.2.1 Experiment setup

The scenarios we tested on were adapted from the work of Fang et al. [20], which presented the full encoding solution for finding chance-constrained static policies. These model the logistics of an urban car-sharing company, where they have a fleet of cars available for short-term rentals within within city limits. Each car may be reserved by different customers in sequence, and each customer may take the car to a series of destinations.

The only requirement is that the cars may not be operated after the company's close of business at the end of the day. Additional complexities such as returning cars to designated locations or time limits on rentals are not modeled. Therefore, the structure of the plan is a series of parallel threads, one for each car, with an overall deadline requirement over all threads. This is similar to the lunar construction scenario in Figure 7-2, except that there are no inter-thread constraints.

Each thread is semantically divided into segments corresponding to each customer's reservation. Within a reservation, the activities alternate between driving to a destination and shopping at the destination. Only the driving activities have uncontrollable duration. Thus, the events corresponding to the beginning of driving, or equivalently the end of shopping, are the controllable events that get assigned by our static policy.

Since these scenarios were provided to us by Fang et al., we do not know the exact parameters that were used to generate the intervals bounds and probabilistic durations for the activities, nor do we know how many trial scenarios were generated for each problem size specification. However, we can list the following facts about the data set: A total of 1800 scenarios were generated. Within these, the number of cars ranged from 1 to 20, the number of customers per car from 1 to 5, and the number of destinations per customer between 1 and 3. That means at the upper range of problem size, there are approximately 300 uncontrollable driving activities in the plan. We also know that the chance constraint risk bounds were set at one of 10%, 20%, or 40%.

Like before, the algorithms are all implemented in Common Lisp, and Ipopt is used as the external NLP solver. However, we used an implementation of the conflict-directed algorithm that follows Algorithm 4.1, rather than the two-level architecture in Chapter 5.

The effect on runtime should be negligible, since the bulk of the time is spent in the NLP solver anyway. In this case, the scenarios were run on a first-generation desktop Core i7.

7.2.2 Results

We compare our conflict-directed algorithm for generating static policies against sending the full encoding to the NLP solver directly. We also evaluate a variant of the full encoding where we turn the reformulated chance constraint into an objective function of minimizing the total risk allocated. The runtimes for the each scenario are plotted in Figures 7-7 through 7-9. Since a wide range of runtimes are encompassed at each problem size, and the ranges for the different algorithms overlap significantly, we introduce the runtime results one algorithm at a time. First, we show the runtimes for the full encoding, in orange squares. Then, we overlay the results for our conflict-directed approach, in blue diamonds. The last overlay is for the full encoding that minimizes risk, in green triangles.

It turns out we have complete runtimes only for the conflict-directed algorithm. For the chance-constrained and risk-minimizing variants of the full encoding approach, the raw data contains runtimes for only approximately the first 1300 and the first 800 scenarios, respectively. Beyond taking too much time to complete, the reasons for the cutoff are unknown. However, the trends are still clear enough for us to draw conclusions.

The main comparison is between the orange and the blue clusters. For small problems, they have comparable runtimes, but the orange cluster grows faster, and by 100 to 150 uncontrollable activities, it clearly averages at least 10 times longer. Both clusters sweep out a wide vertical range for any given problem size. This can be attributed to the Ipopt's own variation in runtime for similar problem sizes. Even though the solver's algorithm is deterministic, it can perform wildly differently for similar-sized inputs.

For the full encoding, which calls the NLP solver only once, we reach runtimes ranging from about 20 minutes to a few hours for plans containing around 150 probabilistic durations. In contrast, the blue diamonds don't reach those extreme runtimes until the plan size increases to about 300 such durations. Even then, running for more than an hour is outlier behavior, and most of the results are clustered around just over a minute to several minutes.

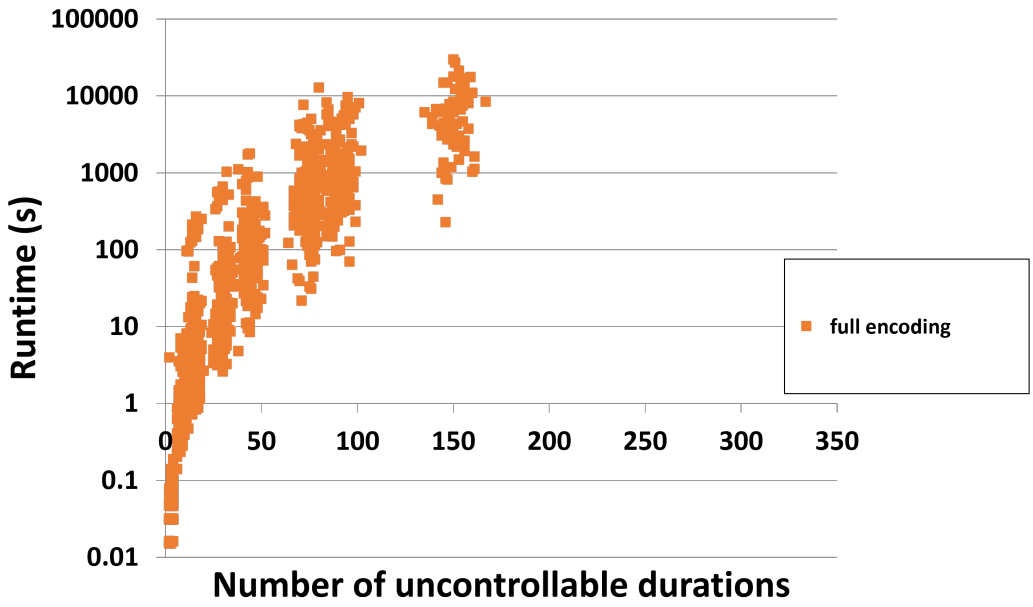

Figure 7-7: When using the full encoding, the solver spends between several minutes to a few hours on plans containing approximately 150 probabilistic durations.

Clearly, our conflict-directed approach can solve problems twice as large at still a faster pace.

The wider overall range of runtimes when using conflicts may be partially attributed to the fact that we have to run the NLP solver multiple times. However, as with the lunar construction scenarios, we found that very few iterations were required to reach an answer: never more than 5. Confoundingly, the scenarios with the longest runtimes never ran for more than two iterations. This means the vast variation in runtime is still primarily due to Ipopt's performance. It might have been the case, for instance, that a particularly large negative cycle with many edges was discovered, resulting in a linear resolution constraint that contained many risk allocation variables. It is possible that such a constraint would be more difficult for Ipopt to process, compared to those in the full encoding. While the full encoding contains many linear constraints, each only involves two event variables and at most one risk allocation variable.

Finally in Figure 7-9, we overlay the runtimes for the risk-minimizing variant of the full encoding. Compared to Figure 7-7, these closely track the runtimes of the chance-constrained version, but end up slightly faster on average. A similar relationship was

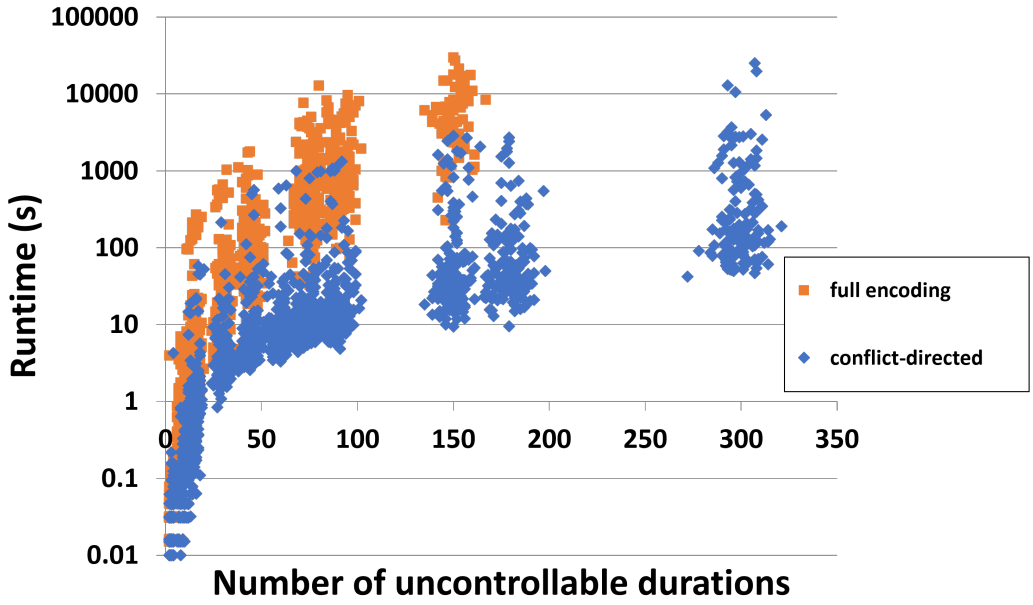

Figure 7-8: Compared to the full encoding, our conflict-directed approach reaches the range of maximum runtimes at about twice the problem size, with 300 probabilistic durations. On average, for the same problem size, our approach's runtime is about 10 times shorter.

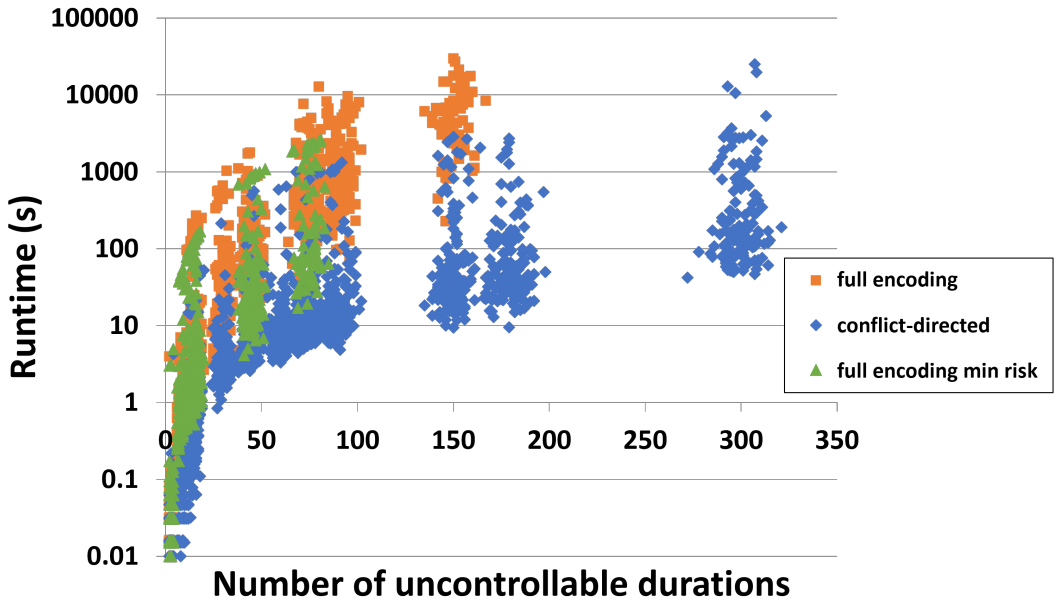

Figure 7-9: Adding a minimum risk objective to the full encoding actually speeds it up slightly.

observed in Fang et al.'s work [20], even though they used a different NLP solver, Snopt [25]. The exact reason for this discrepancy is not clear, as it depends on the internal machinery for handling objective functions versus constraints. It appears that the gradient imposed by an objective function might provide more "incentive" towards approaching an optimal point, if any, whereas as a chance constraint, it's harder to nudge the solver into whatever feasible region there may be.

The last observation is that the shape of each cluster suggests a polynomial-time complexity of our algorithms with respect to problem size. This is true of the full encodings, as Ipopt uses an interior-point method, which runs in polynomial time in terms of problem size and solution accuracy [73]. Technically, our conflict-directed algorithm runs in exponential time, because there could be an exponential number of negative cycles discovered. However, this is the theoretical extreme, as as the results show, the actual number of iterations tends to be very low. Combined with the fact that our conflict-directed approach doesn't need to encode event variables, its time complexity in practice is likely a lower-order polynomial than that of the full encodings.

Chapter 8

Conclusion and Future Work

This thesis has provided the capability to solve risk-bounded scheduling problems using dynamic execution. We've argued that humans intuitively perform such reasoning, but only on scenarios of limited complexity, and without precise quantification of risk. We now offer a way to scale that reasoning to large, complex scenarios that would have previously eluded efforts to be reliably scheduled. Our ultimate contribution is a conflict-directed algorithmic architecture that offers efficiency without sacrificing bounded-risk guarantees. Moreover, we illuminate the theoretical link between solving for chance-constrained static policies and solving for those that are dynamic.

This chapter summarizes our contributions towards understanding the problem and developing a solution. Then, we close by outlining several avenues for extending this work. The most salient one is to be able to specify multiple chance constraints, each over *subsets* of the temporal requirements. This allows us to designate portions of the plan to have higher importance, which is a common need. Second, we consider the ability to efficiently adjust computed policies in response to incremental plan updates. This is called incremental scheduling, and it enhances the usefulness of scheduling algorithms when employed by planners. Finally, we note methods for computing optimal scheduling policies and for approximating the chance constraint. The latter extension sacrifices a small amount of solution space for potentially large gains in runtime.

8.1 Thesis summary

The overall advance of this thesis is to enable dynamic scheduling of temporal plans with bounded-risk guarantees. Previous work in chance-constrained scheduling only offered static schedules. Conversely, STNU models of scheduling have well-developed theory for dynamic execution, but lack any notion of risk. This work can thus be seen as unifying these two branches into a new and rich capability.

The first step in achieving this is to understand the structure and conditions of the problem. Chapter 2 gave us several advances in this regard. First, the pSTN model was defined by transforming the uncontrollable durations of STNUs into probabilistic durations. To encompass arbitrary distributions and correlations for these durations, we defined a joint distribution across the entire space of uncontrollable outcomes. Second, we adopted Hunsberger's notion of a STNU dynamic policy, and applied it to the pSTN outcome space. This gives us clear semantics for executing dynamic scheduling policies on a pSTN, *and* combined with the first advance, we now have likelihood densities attached to the execution histories produced by those policies.

Finally, we separate out the notion of a temporal requirement from duration models in the temporal network. This leads to a straightforward definition of the chance constraint, which imposes a maximum likelihood of failing to satisfy the requirements. Again, the semantics of this are very clear, because we can directly evaluate whether any given execution history satisfies the requirements.

With the problem well-defined, we begin to build a tractable solution method in Chapter 3. Just as we did for chance-constrained static scheduling, and consistent with virtually every solution approach in the literature, we start by reformulating the problem into a deterministic form via the concept of risk allocation. The intuition is to reduce the pSTN into an STNU by chopping off the tails of the probability distributions, while making sure the tail masses, aggregated via the union bound, don't exceed the chance constraint's risk bound. This isolates the problem's probabilistic condition, and allows us to directly leverage STNU scheduling theory to address the remaining temporal requirements and duration models. Any policy for the STNU will be a feasible solution for the original cc-pSTN, provided it is

properly adapted to handle pSTN outcomes.

It is important to note that this decoupling introduces conservatism and reduces the solution space with respect to the original problem. Previous works acknowledged this, but this thesis is the first to rigorously account for the sources of incompleteness. Namely, we explicitly map out the relationships between the policy space, the duration outcome space, and the risk allocation space. With the exception of pathological scenarios, we argued that the losses are likely minimal, and our analysis suggests a future roadmap for empirically measuring the degree of conservatism.

Our algorithmic approach in Chapter 4 then takes advantage of the problem reformulation by employing a hybrid approach, involving two subsolvers. First, we use an NLP solver to process the reformulated chance constraint, giving us a risk allocation. Then, we check the implied STNU for controllability. If it's not controllable, we receive a negative cycle conflict, whose resolution becomes an obstacle in the risk allocation space. Thus, we iteratively add obstacles to our risk allocation space until we either find a controllable STNU – and hence a policy along with it – or we've completely blocked out the chance constraint feasible region.

To obtain dynamic policies, we must check for STNU dynamic controllability and receive DC conflicts back. The difficulty is that unlike SC conflicts, which form half-volume obstacles bounded by linear hyperplanes, DC conflicts form convex polytopes. Our strategy for resolving them is to branch on the polytopes' facets, each of which is a linear hyperplane. Thus, by trying all combinations of one facet from each polytope, we reduce our problem of generating dynamic policies into a series of problems that the static policy algorithm can handle.

In Chapter 5, we recognized that this reduction shares common principles with the conflict-directed approach presented in Chapter 4. Before, we were cutting down on the risk allocation space by solving a series of STNU controllability problems. Now, we encapsulate that in a black box, so that we can cut down on the discrete search space of linear resolutions to DC conflicts. As we scan through the leaves of the search tree, we solve a series of risk allocation problems, using that black box's functionality.

This insight led us to a common architecture for producing both static and dynamic

chance-constrained policies. The static solution comprises two layers interacting in the traditional master-subproblem fashion, while the dynamic solution simply requires a third layer on top to handle the combinatorial search. The middle layer acts as a passthrough for the bottom layer's SRNC DC conflicts, and also optionally returns its own risk allocation conflicts for the third layer to resolve.

Finally in Chapter 7, we provided empirical evidence of the advantages of chance-constrained dynamic policies when scheduling large plans. The primary result is that solving dynamic policies, either with variable or uniform risk allocations, enables significantly more scenarios to be scheduled. Compared to static policies, dynamic policies are shown to solve problems at least 10 times larger at the same rate of success. We also observe that we can obtain dynamic policies by expending only slightly more computational effort than that needed for generating static policies. Thus, this is very strong validation of advancing chance-constrained scheduling from static to dynamic policies.

Lastly, we demonstrate that our conflict-directed approach for static policies is typically 10 times faster than solving a full encoding. While it was impractical for us to implement a full encoding for chance-constrained dynamic policies, the fact that our dynamic policy algorithm had similar runtimes as our static algorithm strongly suggests that we would see similar or greater gains in the dynamic case.

8.2 Multiple chance constraints

We formulated our main problem, Problem 2.20, with a single chance constraint that applies to all the requirements of a pSTN. It is natural to consider, though, defining multiple chance constraints, each over a chosen subset of the requirements. This would allow us to prioritize the likelihood of satisfying the certain requirements, or bundles of them, over others. There is precedent for such prioritization in other chance-constrained contexts, such as path planning For example, Blackmore [7] and Ono [46] present scenarios where remaining in a designated "safe zone" throughout the path is more important than visiting certain waypoints along the way.

In our spaghetti example from Figure 2-5, perhaps our recipe's most important step is

to not let the pasta sit for too long after being strained, or else the strands begin to clump together. Therefore, we might apply the 2% chance constraint to that $[0, 2]$ requirement only, while specifying a more relaxed 5% chance constraint over the remaining requirements. From the user's perspective, this means the other deadlines are more likely to fail than that particular one. So we (and we hope the customer, too) are willing to sacrifice a bit of timely service to avoid compromising the integrity of our recipe.

From our algorithm's perspective, multiple chance constraints are also advantageous in increasing the solution space of policies. In our example, the 2% risk bound needs to be allocated now *only* to any probabilistic durations that could influence the satisfation of the $[0, 2]$ requirement after straining the spaghetti. Intuitively, we can deduce that only the durations outcomes of the boiling spaghetti and heating sauce activities could threaten it. This is because the requirement is defined on a wait activity that represents the merging of the boiling thread with the heating thread. However, the bake meatballs activity's duration has no effect on this $[0, 2]$ requirement, because that happens on a third thread that is still independent by the time these two have merged.

What this illustrates is that by specifying a chance constraint on a local portion of the plan, we don't have to allocate its risk bound to non-relevant activities. Thus, we can be more aggressive in removing the distribution tails on the relevant activities. This leads to an STNU with less uncertainty, and is therefore more likely to admit a scheduling policy. Meanwhile, the additional 5% chance constraint also relaxes our problem. Since it involves the overall deadline, we can see that all three probabilistic durations are included in its risk allocation. However, previously we had to allocate 2% risk over all three durations, so this is still a looser condition.

To adapt our algorithm to handle multiple chance constraints, then, the key additional reasoning we need is to identify the relevant durations for allocating each risk bound. That is, based on the semantics of pSTN dynamic execution according to any policy, which probabilistic durations could affect the satisfaction of that requirement? This is where the value of distinguishing requirements from activities becomes apparent. Based on the semantics of pSTN dynamic execution, we can evaluate whether a requirement is satisfied or not by the time that both events it relates have been executed. It follows that we should

trace the activity threads that led to those events being executed, and collect the probabilistic durations along the way.

There is a nuance, though, and hence why fully handling multiple chance constraints requires additional treatment. In our example, suppose that prior to the plan's start event, we inserted an additional thread of "preparation" activities, such as taking the meatballs out of the freezer and waiting for them to thaw. If that thread had probabilistic durations, then those should *not* be included when allocation risk for our 2% chance constraint after straining. This is because the boiling and heating threads that our $[0, 2]$ requirement depends on were *forked simultaneously* from the original start event of the plan. Thus, anything before that start event is also irrelevant to satisfying the requirement in question. Since plan in general can have arbitrary forking and merging of threads, designing an algorithm that accurately identifies the relevant activities is nontrivial.

To close this discussion, we note that for large plans, defining local chance constraints is extremely advantageous, because it allows us to have a notion of "rolling risk" rather than global risk for the entire plan duration. We saw this in our experiments in Section 7.1, where the portion of scenarios solved dropped as the plan size increased. In large part, this was because we were trying to allocation the same 10% risk bound over increasingly many probabilistic durations. If we had local chance constraints, then we would have been able to maintain the same *rate* of risk as we continue to install additional satellite dishes. Then, any drop in the portion of scenarios solved would be due to the additional ordering restrictions as we add more astronauts to the team.

8.3 Incremental scheduling

When planners make use of scheduling algorithms, they usually step through several candidate plans, and call the scheduler on each. For instance, this strategy is used by both the Kirk [52] and tBurton [70] [71] planners. In such contexts, the candidate plans represent alternative arrangements of activities to achieve the planning goals. These alternatives usually differ from each other only in a few portions, and they share the same structure in the rest.

This can be seen in the scenarios from Section 1.1. For the movie example, we are deciding between two options. However, the choice doesn't happen until we reach Park Street station, when we can either wait for the transfer, or make the rest of the trip on foot. Either way, we still have to take the Red Line from Kendall to Park Street. In the oceanography scenario, our nominal plan may be to chase after the AUV first. But we also have to consider the alternative plan where we stop chasing after the tether runs out, and we have to switch to recovering the ROV first.

Such use cases demonstrate that when the planner calls the scheduler on a modified plan, it would be valuable for the scheduler to consider the shared structure, rather than compute a solution entirely from scratch again. This is known as *incremental* scheduling, in the sense that the input to the scheduling problem is a modification to an existing plan, rather than an entirely new plan. It is also the case that sometimes plans need to modified *during execution*, due to unexpected disturbances or new information. Again, the modifications are usually localized, so incremental recalculation of the policy would be desired.

Efficient incremental scheduling solutions already exist for both STNs [52] and STNUs [51] [45]. Therefore, it is natural to try to extend that capability to pSTNs as well. Our cc-pSTN scheduling solution already relies on existing non-incremental algorithms for checking STNU controllability, so there might be value in swapping in the incremental variants. However, we handle the chance constraint by solving an NLP, which is entirely absent from STN and STNU algorithms. Thus, we would have to develop an incremental strategy to avoid constructing the NLP from scratch.

For insight, we look to how existing incremental scheduling solutions work [9] [24]. When checking STNs for consistency, the underlying data structure is a single-source shortest-paths (SSSP) tree constructed by Bellman-Ford. When an STN constraint is tightened, its effects are only felt downstream. Conversely, when a constraint is loosened or relaxed, it affects the SSSP computation only if it was part of a shortest path. In either case, we can derive a set of rules for invalidating certain portions of the SSSP tree and recomputing just those portions. Incremental dynamic controllability for STNUs operates on similar principles, just with slightly more complex rules to determine invalidation [51].

For cc-pSTNs, then, the question for incremental solving is what portions of the internal

NLP should get invalidated. Recall that the NLP consists of a reformulated chance constraint plus a collection of (selected) linear constraints, representing cycles or paths in the implied STNU's distance graph. So in brief, the answer is that when a pSTN constraint gets tightened, if that constraint corresponds to a distance graph edge that participates in one of those linear constraints, then that linear constraint gets tightened as well, so it is safe to keep it in the NLP. However, if a pSTN constraint is removed, then its associated distance graph edges no longer exist. Therefore, any linear constraints that rely on those edges should be removed from the NLP.

This is only a broad overview, and there are edge cases to consider. However, once the main principles are understood, it seems the solution would be relatively straightforward. As a final point, we note that using a black box NLP solver forces us to reinitialize its input every time we call it. If there were a way to call the solver with "warm starts", or to reuse its internal data structures, we might realize further incremental gains. This would also be useful for our current non-incremental algorithm, since we are typically only adding new linear constraints or swapping a few out between calls.

8.4 Optimization and approximation

The last two extensions we present have already been developed for chance-constrained static policies. In this section, we summarize how they work, and how one might adapt them to our algorithm for dynamic policies.

The first extension is to produce not just any policy that satisfies the chance constraint, but an *optimal* one with respect to some objective function. For instance, a common objective is to minimize the total duration of a plan, known as the makespace. Considering our restaurant spaghetti example, instead of having an overall 15 minute deadline requirement, we could turn it into a $[0, D]$ constraint, where D is a variable representing the makespan, and try to minimize D in our solutions.[1] Keeping the $[0, 15]$ requirement would be redundant, because whatever value we find for D in the solution, that determines whether we would have been able to meet the 15 minute deadline.

[1]The logical evolution of your restaurant's signature dish is to mass-produce it as a frozen dinner. Your

We could also set objectives on the internal coordinating throughout a plan. In our lunar construction scenario, we could insert extra activities that model the idle times as astronauts queue up to confirm their satellite dish installations, and specify their sum total as the objective to minimize. Finally, we could always elect to minimize total risk, instead of bounding it by a chance constraint. However, we could still have local chance constraints that enforce minimum levels of safety (i.e., maximum acceptable risk) on certain requirements.

Optimal static policies were first demonstrated by Fang [20] in the full encoding approach. Since it calls the NLP solver just once, and such solvers already accept objective functions, it was natural to include an objective with the compiled constraints. Subsequent work by Wang [69] focused on developing the conflict-directed approach for *satisfying* the chance constraint, and did not include an objective.

Recently, though, optimization has been incorporated into the latter approach [68]. The key result is that rather than include the objective function on every call to the NLP solver, include only once a feasible solution has been found. In other words, run the original conflict-directed algorithm, but rather than exit when a feasible solution is found, *continue* to perform rounds of risk allocation and conflict discovery, now with the objective included, until no more conflicts are discovered. This strategy allows us to converge to a feasible solution quickly, without wasting energy upfront optimizing in regions of the risk allocation space that are likely to be cut out anyway.

Adding an objective function to the NLP works well in the static case, because the constraint program only ever collects additional linear constraints. In the dynamic case, the NLP is partially governed by the linear disjuncts selected by Level 3's branching. Thus, when Level 3 backtracks, constraints in the NLP can be swapped out for others, and feasibility is no longer guaranteed. Therefore, when incorporating an objective function into our dynamic policy algorithm, unless we include it with every call to the NLP solver, there are nuances in deciding when to include it, and that is the topic of futurer work.

The second extension addresses our current need to call an NLP solver, which we've noted is by far the dominant factor in runtime. Santana derived a linear overapproximation of the risk taken by removing the tails of durations' distributions [50]. This allowed

job is now to churn out as many of these as you can, which means executing the recipe as fast as possible.

him to rewrite the reformulated chance constraint as a tighter linear constraint, and hence transform the NLP into a pure LP. LP solvers are generally much faster than NLP solvers, so that even though Santana implemented the full encoding with this linearly approximated chance constraint, it still outperformed the Wang's conflict-directed approach by typically two orders of magnitude.

Unlike an objective function, which is imposed on top of the constraints in our constraint program, the reformulated chance constraint is an essential component of every call to the solver. Thus, employing Santana's linear approximation in our conflict-directed hybrid algorithm would be a straightforward replacement in both the static and dynamic cases. We could expect similar improvements in runtime, which would be valuable for solving even larger problem instances. There are a couple limitations to be aware of, though.

First, the method requires the probability density function (PDF) of each duration's distribution to be unimodal, and we've already argued on page 64 that this is typically a very reasonable modeling assumption. The approximation itself works by turning vertical slices of the PDF into thin rectangles. Having a unique mode (or even a plateau) ensures that rectangles on either side of it will cover the slices' entire areas. The linearized constraint is then formed with variables specifying where the slicings occur. It follows that the more slicing variables we include, the tighter the approximation.

The two tradeoffs, then, are that we replace the NLP with an LP, while likely introducing many more auxiliary slicing variables, and also increasing conservatism due to the overapproximation. We can reduce the conservatism by adding more slices, but we'll get diminishing returns, and eventually overwhelm the LP. These tradeoffs should be studied empirically, so that users of this method can have an expectation of how to calibrate their linearized approximation.

Bibliography

[1] Jordan R. Abrahams, David A. Chu, Grace Diehl, Marina Knittel, Judy Lin, William Lloyd, James C. Boerkoel, Jr., and Frank Jeremy. DREAM: an algorithm for mitigating the overhead of robust rescheduling. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 3–12. AAAI Press, 2019.

[2] Shyan Akmal, Savana Ammons, Hemeng Li, and James C. Boerkoel Jr. Quantifying degrees of controllability in temporal networks with uncertainty. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 22–30. AAAI Press, 2019.

[3] Egon Balas. Disjunctive programming and a hierarchy of relaxations for discrete optimization problems. *SIAM Journal on Algebraic Discrete Methods*, 6(3):466–486, 1985.

[4] Jacques F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4(1):238–252, 1962.

[5] Nikhil Bhargava. *Multi-Agent Coordination under Limited Communication*. PhD thesis, Massachusetts Institute of Technology, 2020.

[6] Nikhil Bhargava, Cameron W. Pittman, and Brian C. Williams. Representing uncertain communication in temporal networks. Submitted to the Journal of Artificial Intelligence Research.

[7] Lars Blackmore. *Robust Execution for Stochastic Hybrid Systems*. PhD thesis, Massachusetts Institute of Technology, 2007.

[8] Lars Blackmore and Masahiro Ono. Convex chance constrained predictive control without sampling. In *AIAA Guidance, Navigation, and Control Conference*, page 5876, 2009.

[9] Amedeo Cesta and Angelo Oddi. Gaining efficiency and flexibility in the simple temporal problem. In *Proceedings of the Third International Workshop on Temporal Representation and Reasoning (TIME)*, pages 45–50. IEEE Computer Society, 1996.

[10] Boris V. Cherkassky and Andrew V. Goldberg. Negative-cycle detection algorithms. *Mathematical Programming*, 85(2):277–311, 1999.

[11] John W. Chinneck. Finding a useful subset of constraints for analysis in an infeasible linear program. *INFORMS Journal on Computing*, 9(2):164–174, 1997.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[13] Jing Cui, Peng Yu, Cheng Fang, Patrik Haslum, and Brian C. Williams. Optimising bounds in simple temporal networks with uncertainty under dynamic controllability constraints. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 52–60. AAAI Press, 2015.

[14] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[15] Leonardo Mendonça de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In *Proceedings of the Nineteenth International Conference on Computer Aided Verification (CAV)*, pages 20–36. Springer, 2007.

[16] Rina Dechter. *Constraint processing*. Elsevier Morgan Kaufmann, 2003.

[17] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1):61–95, 1991.

[18] Jack Ewing and Patricia Cohen. How car shortages are putting the world's economy at risk. `https://www.nytimes.com/2021/11/02/business/car-shortage-global-economy.html`, 2021.

[19] Cheng Fang. *Efficient algorithms and representations for chance-constrained mixed constraint programming*. PhD thesis, Massachusetts Institute of Technology, 2021.

[20] Cheng Fang, Peng Yu, and Brian C. Williams. Chance-constrained probabilistic simple temporal problems. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2264–2270. AAAI Press, 2014.

[21] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): fast decision procedures. In *Proceedings of the Sixteenth International Conference on Computer Aided Verification (CAV)*, pages 175–188. Springer, 2004.

[22] Michael Gao, Lindsay Popowski, and James C. Boerkoel, Jr. Dynamic control of probabilistic simple temporal networks. In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence*, pages 9851–9858. AAAI Press, 2020.

[23] Arthur M. Geoffrion. Generalized benders decomposition. *Journal of Optimization Theory and Applications*, 10(4):237–260, 1972.

[24] Alfonso Gerevini, Anna Perini, and Francesco Ricci. Incremental algorithms for managing temporal constraints. In *Proceedings of the Eigth International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 360–365. IEEE Computer Society, 1996.

[25] Philip E. Gill, Walter Murray, and Michael A. Saunders. SNOPT: an SQP algorithm for large-scale constrained optimization. *SIAM Review*, 47(1):99–131, 2005.

[26] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[27] Olivier Guieu and John W. Chinneck. Analyzing infeasible mixed-integer and integer linear programs. *INFORMS Journal on Computing*, 11(1):63–77, 1999.

[28] John N. Hooker and María Auxilio Osorio Lama. Mixed logical-linear programming. *Discrete Applied Mathematics*, 96-97:395–442, 1999.

[29] John N. Hooker and Greger Ottosson. Logic-based benders decomposition. *Mathematical Programming*, 96(1):33–60, 2003.

[30] Luke Hunsberger. Fixing the semantics for dynamic controllability and providing a more practical characterization of dynamic execution strategies. In *Proceedings of the Sixteenth International Symposium on Temporal Representation and Reasoning (TIME)*, pages 155–162. IEEE Computer Society, 2009.

[31] Luke Hunsberger. A fast incremental algorithm for managing the execution of dynamically controllable temporal networks. In *Proceedings of the Seventeenth International Symposium on Temporal Representation (TIME)*, pages 121–128. IEEE Computer Society, 2010.

[32] Luke Hunsberger. New techniques for checking dynamic controllability of simple temporal networks with uncertainty. In *Revised Selected Papers from the Sixth International Conference on Agents and Artificial Intelligence (ICAART)*, volume 8946 of *Lecture Notes in Computer Science*, pages 170–193. Springer, 2014.

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

[34] Raj Krishnan. Solving hybrid decision-control problems through conflict-directed branch & bound. Master's thesis, Massachusetts Institute of Technology, 2004.

[35] Hoong Chuin Lau, Jia Li, and Roland H. C. Yap. Robust controllability of temporal constraint networks under uncertainty. In *Proceedings of Eighteenth IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 288–296. IEEE Computer Society, 2006.

[36] Hui X. Li. Generalized conflict learning for hybrid discrete/linear optimization. Master's thesis, Massachusetts Institute of Technology, 2006.

[37] Kyle Lund, Sam Dietrich, Scott Chow, and James C. Boerkoel, Jr. Robust execution of probabilistic temporal plans. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pages 3597–3604. AAAI Press, 2017.

[38] Paul H. Morris. A structural characterization of temporal dynamic controllability. In *Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 4204 of *Lecture Notes in Computer Science*, pages 375–389. Springer, 2006.

[39] Paul H. Morris. Dynamic controllability and dispatchability relationships. In *Proceedings of the Eleventh International Conference on the Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*, volume 8451 of *Lecture Notes in Computer Science*, pages 464–479. Springer, 2014.

[40] Paul H. Morris. The mathematics of dispatchability revisited. In *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 244–252. AAAI Press, 2016.

[41] Paul H. Morris and Nicola Muscettola. Temporal dynamic controllability revisited. In *Proceedings of the Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*, pages 1193–1198, 2005.

[42] Paul H. Morris, Nicola Muscettola, and Thierry Vidal. Dynamic control of plans with temporal uncertainty. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 494–502, 2001.

[43] Nicola Muscettola, Paul H. Morris, and Ioannis Tsamardinos. Reformulating temporal plans for efficient execution. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 444–452. Morgan Kaufmann, 1998.

[44] Mikael Nilsson, Jonas Kvarnström, and Patrick Doherty. Incremental dynamic controllability revisited. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI Press, 2013.

[45] Mikael Nilsson, Jonas Kvarnström, and Patrick Doherty. Efficient processing of simple temporal networks with uncertainty: algorithms for dynamic controllability verification. *Acta Informatica*, 53(6–8):723–752, 2016.

[46] Masahiro Ono. *Robust, Goal-directed Plan Execution with Bounded Risk*. PhD thesis, Massachusetts Institute of Technology, 2011.

[47] Masahiro Ono and Brian C. Williams. An efficient motion planning algorithm for stochastic dynamic systems with constraints on probability of failure. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 1376–1382, 2008.

[48] Masahiro Ono, Brian C. Williams, and Lars Blackmore. Probabilistic planning for continuous dynamic systems under bounded risk. *Journal of Artificial Intelligence Research*, 46:511–577, 2013.

[49] Cameron W. Pittman. Delay STNU benchmarks. `https://gitlab.com/mit-mers/delay-stnu-benchmarks`. Accessed: 2022-06.

[50] Pedro Santana, Tiago Vaquero, Cláudio Toledo, Andrew J. Wang, Cheng Fang, and Brian Williams. PARIS: A polynomial-time, risk-sensitive scheduling algorithm for probabilistic simple temporal networks with uncertainty. In *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 267–275. AAAI Press, 2016.

[51] Julie A. Shah, John Stedl, Brian C. Williams, and Paul Robertson. A fast incremental algorithm for maintaining dispatchability of partially controllable plans. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 296–303, 2007.

[52] I-Hsiang Shu, Robert T. Effinger, and Brian C. Williams. Enabling fast flexible planning through incremental temporal reasoning with conflict extraction. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 252–261. AAAI, 2005.

[53] Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[54] John L. Stedl. Managing temporal uncertainty under limited communication: A formal model of tight and loose team coordination. Master's thesis, Massachusetts Institute of Technology, 2004.

[55] Peter Stone and Manuela M. Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous. Robots*, 8(3):345–383, 2000.

[56] Eric Timmons and Brian C. Williams. Best-first enumeration based on bounding conflicts, and its application to large-scale hybrid estimation. *Journal of Artificial Intelligence Research*, 67:1–34, 2020.

[57] Ioannis Tsamardinos. A probabilistic approach to robust execution of temporal plans with uncertainty. In *Proceedings of the Second Hellenic Conference on AI (SETN)*, volume 2308 of *Lecture Notes in Computer Science*, pages 97–108. Springer, 2002.

[58] Ioannis Tsamardinos, Nicola Muscettola, and Paul H. Morris. Fast transformation of temporal plans for efficient execution. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference (AAAI) (IAAI)*, pages 254–261. AAAI Press / The MIT Press, 1998.

[59] Ioannis Tsamardinos, Martha E. Pollack, and Sailesh Ramakrishnan. Assessing the probability of legal execution of plans with temporal uncertainty. In *Proceedings of the ICAPS 2003 Workshop on Planning under Uncertainty and Incomplete Information*, 2003.

[60] Thierry Vidal and Hélène Fargier. Handling contingency in temporal constraint networks: From consistency to controllabilities. *Journal of Experimental & Theoretical Artificial Intelligence*, 11(1):23–45, 1999.

[61] Thierry Vidal and Malik Ghallab. Dealing with uncertain durations in temporal constraint networks dedicated to planning. In *Proceedings of the Twelfth European Conference on Artificial Intelligence (ECAI)*, pages 48–54, 1996.

[62] Helwing Villamizar. Boeing confirms 777X deliveries for 2025. `https://airwaysmag.com/boeing-777x-delayed-2025/`, 2022.

[63] Andreas Wächter. Short tutorial: Getting started with ipopt in 90 minutes. In *Combinatorial Scientific Computing*, volume 09061 of *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl, 2009.

[64] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.

[65] Benjamin W. Wah and Dong Xin. Optimization of bounds in temporal flexible planning with dynamic controllability. In *Proceedings of the Sixteenth IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 40–48. IEEE Computer Society, 2004.

[66] Benjamin W. Wah and Dong Xin. Optimization of bounds in temporal flexible plans with dynamic controllability. *International Journal on Artificial Intelligence Tools*, 16(1):17–44, 2007.

[67] Andrew J. Wang. Risk allocation for temporal risk assessment. Master's thesis, Massachusetts Institute of Technology, 2013.

[68] Andrew J. Wang, Cheng Fang, and Brian C. Williams. Chance-constrained static schedules for temporally probabilistic plans. Accepted by the Journal of Artificial Intelligence Research.

[69] Andrew J. Wang and Brian C. Williams. Chance-constrained scheduling via conflict-directed risk allocation. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 3620–3627. AAAI Press, 2015.

[70] David Wang and Brian C. Williams. tBurton: A divide and conquer temporal planner. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 3409–3417. AAAI Press, 2015.

[71] David C. Wang. *A Factored Planner for the Temporal Coordination of Autonomous Systems*. PhD thesis, Massachusetts Institute of Technology, 2015.

[72] Brian C. Williams and Robert J. Ragno. Conflict-directed A^* and its role in model-based embedded systems. *Discrete Applied Mathematics*, 155(12):1562–1595, 2007.

[73] Margaret Wright. The interior-point revolution in optimization: History, recent developments, and lasting consequences. *Bulletin of the American Mathematical Society*, 42(1):39–56, 2005.

[74] Peng Yu. *Collaborative Diagnosis of Over-Subscribed Temporal Plans*. PhD thesis, Massachusetts Institute of Technology, 2016.

[75] Peng Yu, Cheng Fang, and Brian C. Williams. Resolving uncontrollable conditional temporal problems using continuous relaxations. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI Press, 2014.

[76] Peng Yu, Cheng Fang, and Brian C. Williams. Resolving over-constrained probabilistic temporal problems through chance constraint relaxation. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. AAAI Press, 2015.

[77] Peng Yu and Brian C. Williams. Continuously relaxing over-constrained conditional temporal problems through generalized conflict learning and resolution. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI)*, 2013.