Flexible Execution of Plans with Choice and Uncertainty

by

Patrick Raymond Conrad

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2010

Author . . . . . .
Department of Aeronautics and Astronautics
August 19, 2010

Certified by . . . . . .
Brian C. Williams
Professor
Thesis Supervisor

Accepted by . . . . . .
Eytan H. Modiano
Associate Professor of Aeronautics and Astronautics,
Chair, Graduate Program Committee

Flexible Execution of Plans with Choice and Uncertainty

by

Patrick Raymond Conrad

Submitted to the Department of Aeronautics and Astronautics
on August 19, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Dynamic plan execution strategies allow an autonomous agent to respond to uncertainties, while improving robustness and reducing the need for an overly conservative plan. Executives have improved robustness by expanding the types of choices made dynamically, such as selecting alternate methods. However, in some approaches to date, these additional choices often induce significant storage requirements to make flexible execution possible. This paper presents a novel system called Drake, which is able to dramatically reduce the storage requirements in exchange for increased execution time for some computations.

Drake frames a plan as a collection of related Simple Temporal Problems, and executes the plan with a fast dynamic scheduling algorithm. This scheduling algorithm leverages prior work in Assumption-based Truth Maintenance Systems to compactly record and reason over the family of Simple Temporal Problems. We also allow Drake to reason over temporal uncertainty and choices by using prior work in Simple Temporal Problems with Uncertainty, which can guarantee correct execution, regardless of the uncertain outcomes. On randomly generated structured plans with choice, framed as either Temporal Plan Networks or Disjunctive Temporal Problems, we show a reduction in the size of the solution set of around four orders of magnitude, compared to prior art.

Thesis Supervisor: Brian C. Williams
Title: Professor

Acknowledgments

After two years, I finally get to write the last words of this thesis. It has been a long process, but it has been every bit as rewarding as I hoped at the outset. I could not have finished this work without the constant personal and technical support of everyone in the Model-based, Embedded and Robotics Systems group. Specifically, I need to thank Professor Williams for introducing me to model-based autonomy, pointing me in the right direction for this research, and providing guidance along the way.

A special thank you goes to Julie Shah for providing the inspiration behind this work, helping me brainstorm numerous technical ideas, and providing some solutions to me outright. To everyone else who spent time in the group with me, David, Bobby, Shannon, Hui, Hiro, Cristi, Henri, Larry, Alborz, Stephanie, Gustavo, Andrew, Andreas, and Seung, thank you for helping to make the experience exciting, fun, and technically engaging. I am grateful to all of you for your insights and a rather large stack of edited drafts.

Additionally, I could not have finished without my friends and family to keep me going. You will finally stop getting updates on my thesis, for a little while, anyway.

Finally, I would like to thanks my sponsors, NSF and the Department of Defense, for their generous fellowships that supported me during this work.

Contents

Chapter 1

Introduction

Model-based executives strive to elevate the level of programming for autonomous systems to intuitive, goal-directed commands, providing guarantees of correctness. For example, the user can direct a rover to complete a series of activities, to drive and collect samples, and provide temporal constraints on those instructions. Using a model-based executive, a user can provide a specification of the correct behavior of the robot and leave it to a program, the executive, to determine an appropriate course of action that will meet those goals. Engineers are then alleviated of the need to program a specific set of routines for each robot that can meet those requirements at run-time.

Ideally, model-based executives should be reactive to disturbances and faults. One useful strategy for creating executives that are robust to disturbances is to delay decision making until run-time. This allows an executive to make decisions for a later part of a plan with the benefit of knowing what happened during earlier portions of that plan. In contrast, a system that makes all decisions before execution must predict what outcomes might happen and cannot adjust if something unexpected happens. Therefore, delaying decision making and following a strategy of least commitment can improve system robustness, improve guarantees of correctness, and reduce unnecessary conservatism. This manuscript develops Drake, a plan executive that delays the scheduling of events and the selection of discrete choices until run-time.

First, this chapter provides an overview of the desired behavior of the system.

Section 1.2 provides an intuitive overview of Drake's techniques. Finally, Section 1.3 discusses related work.

1.1 Desired Behavior

To begin describing Drake's desired behavior, consider an example of a flexible plan that contains flexible durations and one discrete choice.

Example 1.1 A rover has 100 minutes to work before a scheduled contact with its operators. Before contact, the rover must traverse to the next landmark, taking between 30 and 70 minutes. To fill any remaining time, the rover has two options: collect some samples or charge its batteries. Collecting samples consistently takes 50 to 60 minutes, whereas charging the batteries can be usefully done for any duration up to 50 minutes. □

This work develops Drake, an executive that allows the rover to select, as the plan is executed, among all types of available choices of activities and durations. This plan describes the rover's activities and the temporal requirements placed on them. The length of the actions are represented with intervals to show flexibility in the rover's capabilities and possibly some uncertainty. Drake is able to delay selecting between charging the batteries and collecting samples until it learns whether the drive is short enough to make collecting samples feasible. This dynamic execution style contrasts with planners that commit to a precise course of action and durations at the outset of the plan. Since Drake considers all potential executions and efficiently makes commitments as the execution unfolds, it can often switch plans if something unexpected happens, without a costly re-planning step. Furthermore, we also allow Drake to reason about an explicit model of uncertainty. If we provide Drake with an explicit set of durations that it cannot control, but which are selected by nature, Drake can prove whether or not it is robust to all the possible outcomes.

The appeal of dynamic executives is simple: making decisions later means that more information is available, allowing the executive to react to real-world outcomes

and make decisions with less uncertainty, reducing the conservatism required to guarantee correctness. The challenge, however, is that Drake must make decisions quickly enough to satisfy the demands of real-time execution, while guaranteeing that it does not violate any of the constraints set forward in the original plan. Muscettola showed that the temporal constraint reasoning performed by an on-line executive can be made efficient by a pre-processing step referred to as *compilation* [10]. A dispatcher then uses the compiled form of the problem to make decisions at run-time. Essentially, the compilation step makes explicit the consequences of different courses of action available to the dispatcher, allowing it to swiftly make decisions without a risk of overlooking indirect consequences of the input plan. Furthermore, when uncertainty is explicitly modeled, dynamic executives can correctly execute plans that a static executive cannot [9].

The rover scenario demonstrates an example of an implicit constraint that the executive must reveal to correctly dispatch the plan. The entire plan must complete in under 100 minutes and collecting samples takes at least 50 minutes. This leaves only 50 minutes to drive, but the plan specifies that driving may take 70 minutes. Although Drake could theoretically derive this implicit restriction on the driving duration at run-time, real-time decision making latency can be reduced by requiring that this and all other constraints are explicitly recorded, so that choices can be made without searching the entire plan.

These dispatching techniques depend upon a temporal constraint representation called Simple Temporal Problems (STPs), or Simple Temporal Problems with Uncertainty (STPUs), and a corresponding set of algorithms [4]. These formalisms allow for efficient reasoning about the scheduling of events given temporal constraints and models of uncertainty. However, STPs do not model choices between constraints, which is a crucial aspect of plan descriptions. The most common strategy for reasoning about choices within temporal constraints is to create a family of STPs [16, 11, 19]. Within the family of STPs, there is a *component* STP for each possible choice, which specifies all temporal constraints for that choice. This strategy creates a large expansion in the encoding of the plan. We build on prior work in dynamic dispatching for plans

with choice by introducing a compact representation of the compiled form, making it more tractable for embedded systems in terms of memory usage.

To characterize the benefits of Drake, we implemented it in Lisp and evaluated its performance on randomly generated problems. Note that our implementation is substantively the same as the algorithms presented in this thesis and performs identical computation, but does not organize the computation with the same data structures or functional divisions. Our results show that Drake's compact encoding can reduce the size of the compiled representation by around four orders of magnitude for problems with 10,000 component STPs. Overall, Drake trades off the compact storage for an increase in processing time at run-time and compile-time. The compile-time is often improved by several orders of magnitude, but is occasionally worse by about two orders of magnitude. The run-time latency is typically much worse than prior work, but is generally less than a second for most moderately sized problems, making it feasible for real systems.

The next section presents a broad overview of our method.

1.2 Overview of the Method

Our objective is to develop a system that can dynamically execute plans with choice, represented as families of STPs, or STPUs if there is a model of uncertainty. This section gives an overview of our method by walking through the essential steps of preparing and dynamically executing the problem from Example 1.1. Furthermore, it illustrates the compact representation that underlies this work and provides an intuition for why the representation is compact. For simplicity, we will not include uncertainty in this description.

When the executive makes decisions in real-time, it ensures that every decision it makes satisfies every constraint of the plan. If the proposed decision does not violate any constraints, then it can be acted upon; otherwise, the proposition is discarded. This technique is fast, because such tests are easily performed on the temporal constraints, but unless we are careful about the form of the problem, this

system might incorrectly schedule some events.

Example 1.2 uses the rover example to explain an incorrect execution and how compilation can prevent this problem.

Example 1.2 In the plan described in Example 1.1, all activities must complete in under 100 minutes and collecting samples takes at least 50 minutes. This leaves only 50 minutes to drive, yet driving is listed as having a flexible duration of up to 70 minutes. When making scheduling decisions, a proposed duration of 60 minutes would appear acceptable, even though it is not feasible. Muscettola showed that this issue can be solved by reasoning about the constraint ahead of time, and by modifying the duration to explicitly state that the drive duration can be a maximum of 50 minutes if it intends to collect samples[10]. More generally, the limitation on the driving duration is an *implicit* constraint of the problem. The compilation process must make every *implicit* constraint *explicit* in order to allow efficient dispatching. □

A *dispatchable form* is defined as a reformulation of the plan such that local tests are sufficient to guarantee that the dispatcher, the run-time component of the executive, will not violate any constraints of the input plan during execution. Converting from the original input problem to a dispatchable form is done through a reasoning step called *compilation.* Intuitively, the executive thinks through the consequences of every possible decision during compilation and records those results in a readily accessible way. With these consequences readily available in the dispatchable form, the executive can easily look up whether any given decision is feasible, thus simplifying the task at run-time.

1.2.1 Labeled Distance Graphs and Compilation

In Simple Temporal Problems, which only place fixed bounds on the time elapsed between executing events, creating a dispatchable form is simple. The problem is to make explicit all implicit constraints between events. Reasoning over the constraints can be restated as solving a shortest path problem. The All-Pairs Shortest Path graph of the input STP is the dispatchable form of the problem [10].

Adding discrete choices complicates compilation because the dispatchable form must then include the implications of the temporal decisions and the discrete choices. Recall that in the rover example, the discrete choice is between collecting samples and charging the batteries. Each set of possible discrete choices creates a single *component* STP, which can be dispatched with the standard STP techniques. However, each set of discrete choices has a different STP, which the executive must record. A simple way to consider the consequences of the discrete choices, and the technique adopted by Tsamardinos, is to separately record every combination of discrete choices [19]. This method is easily understood, but inefficient because it assumes that every combination of choices is completely different from all others; however, this is rarely the case.

Example 1.3 Consider what happens when the rover example is split into component STPs, as shown in Figure 1-1. Observe that while the STPs are different, many nodes, representing events, and edges, representing activities, are common to both sub-plans. Most of the events and constraints are directly taken from the original problem, except for the drive duration, which is modified to represent the limitation imposed when collecting samples. □

Although the redundancy between the components STPs in Figure 1-1 is limited because there are only two component STPs, the number of copies can scale exponentially with the number of choices. Furthermore, this redundancy is carried through the compilation and dispatching algorithms, leading to a great deal of redundant work and storage. This work explores the idea of merging the separate component STPs into a single representation. It takes full advantage of their similarities to allow compact storage and efficient reasoning.

The challenge of unifying the two STPs is that we need a lossless compression; we require a way to avoid duplicating the identical elements without losing the power to represent every difference between them. Our solution is to introduce a labeling scheme, inspired by prior work on Assumption-based Truth Maintenance Systems (ATMS); by annotating those edges that correspond to the general problem and those that are specific to sample collecting and charging [2].

Example 1.4 Figure 1-2 gives an informal version of what our labeling scheme looks like. The colors and Greek letters represent the different possible scenarios: blue and α for collecting samples, green and β for charging the batteries, and black and γ for universal constraints. This diagram avoids the repetition of identical constraints without confusing which edges correspond to which choices. Note that the driving edge is replaced with two copies, the original constraint and the tightened one for the sample collecting case. □

After creating the representation of the component STPs, we need to compile them. The figures already show the compiled versions. In Tsamardinos's approach, explicitly enumerating the component STPs, each one is independently compiled. Specifically, Figure 1-1a originally would have had a drive duration of $[30, 70]$, as specified in the original plan. Then the compiler reasons that this duration must be tightened because of the overall deadline of 100 minutes and the restriction that sample collecting takes at least 50 minutes. Figure 1-1a is unchanged from the initial form, because the necessary constraints are already recorded explicitly in the plan.

The labeled version must replicate these reasoning steps. Specifically, it uses edge (A, F), with weight $[0, 100], \gamma$, and the collecting samples duration $[50, 60]\alpha$ to derive the restriction on the drive duration. These two constraints have different labels, so the tightest of them is placed on the new constraint, $[30, 50]\alpha$. This new constraint does not replace the old one, because the existing $[30, 70]\gamma$ covers the charging situation also, which the new one does not. This compiled form directly records all the constraints the dispatcher needs to obey at run-time, as described in the next sub-section.

1.2.2 Dispatching the Labeled Representation

Dispatching using Drake's labeled representation requires updating the STP dispatching algorithm so that it handles the labels; this is a straightforward process. The following example describes a few steps of Drake's dispatching process on this example in order to demonstrate the difference.

Example 1.5 Assume that the start event, A, in Figure 1-2, is executed at $t = 0$. At some later time, $t = 40$, the executive needs to determine if it should execute an event. The event B's predecessor, A has been executed, so it may be executed. The other events have B as a predecessor and must wait for it to execute. At time $t = 40$, Drake considers executing B. This time satisfies both constraints on B's execution, given by edges (A, C), $[30, 70]\gamma$ and $[30, 50]\alpha$. Therefore, B can be executed at $t = 40$ without any consideration of whether the rover will collect samples or charge.

In contrast, if Drake repeated the same decision process at $t = 60$, it would notice that the constraint for collecting samples was violated, because $60 \notin [30, 50]\alpha$. Therefore, collecting samples is no longer possible, and Drake would know that it must charge the batteries and follow all remaining constraints for that option. □

The compact representation provided by the labeling system is suitable for dispatching because it is still possible to determine the implications of all choices, just as it was possible by representing the choices as separate STPs.

This section has provided an intuitive sense of why labels are helpful and suggested a form they might take. This thesis formally develops labels, proves that they indeed offer the lossless compression we seek, and modifies the compilation and dispatching algorithms to use this new formalism. The labeling formalism results in a data structure we call *labeled value sets*, which perform quite well for the linear constraints used in these plans. This efficient representation mechanism provides a clean abstraction, making it relatively simple to update constraint reasoning algorithms in order to work with labeled value sets.

The next section discusses related work and concludes with an outline of this thesis

1.3 Related Work

As a dynamic executive, Drake is derived from prior research on dynamic execution of TPNs and DTPs. Muscettola first proposed the separation of the compiler and dispatcher for efficient run-time reasoning on STPs [10]. Then Morris et al. expanded

the dispatcher to provide guarantees of correctness for STPUs, proving that a dynamic strategy can correctly execute plans that a static strategy cannot [9]. Tsamardinos added choice to the dispatcher, executing DTPs, by expanding the DTP encoding into one STP per choice [19].

Kirk is a dynamic executive for TPNs [8]. Kirk performs optimal method selection just before run-time, assigning the discrete choices and then dispatching the resulting STP with Muscettola's work. If some outcome invalidates the STP that Kirk chose, then Kirk performs a re-planning step, selecting a new STP consistent with the execution thus far. Further research developed incremental techniques to allow Kirk to re-plan with lower latency, making it more feasible for an on-line system [13].

As an executive, Drake is designed to be similar to Kirk, in that it is a dynamic executive for TPNs. However, Drake is differentiated in that it uses a compilation strategy to defer commitment to both event times and discrete choices as long as possible and therefore avoids an explicit re-planning step or Kirk's tentative commitment to choices. Unfortunately, avoiding the re-planning step makes some run-time reasoning slower than the reasoning for dispatching an STP, creating a trade-off between nominal and off-nominal performance that we do not address further in this thesis.

Drake's technique builds from Tsamardinos's approach and leverages ideas from Assumption-based Truth Maintenance Systems (ATMS) to make the representation more compact [2]. Drake derives an interesting feature from its heritage in ATMSs, that it is simple to reason about hypothetical choices, because the consequences of all possible choices are explicitly and compactly represented. In contrast, Kirk is guaranteed to find another plan if one exists, but does not track what other possibilities might exist. Future work may be able to exploit these hypothetical situations to add capabilities to Drake.

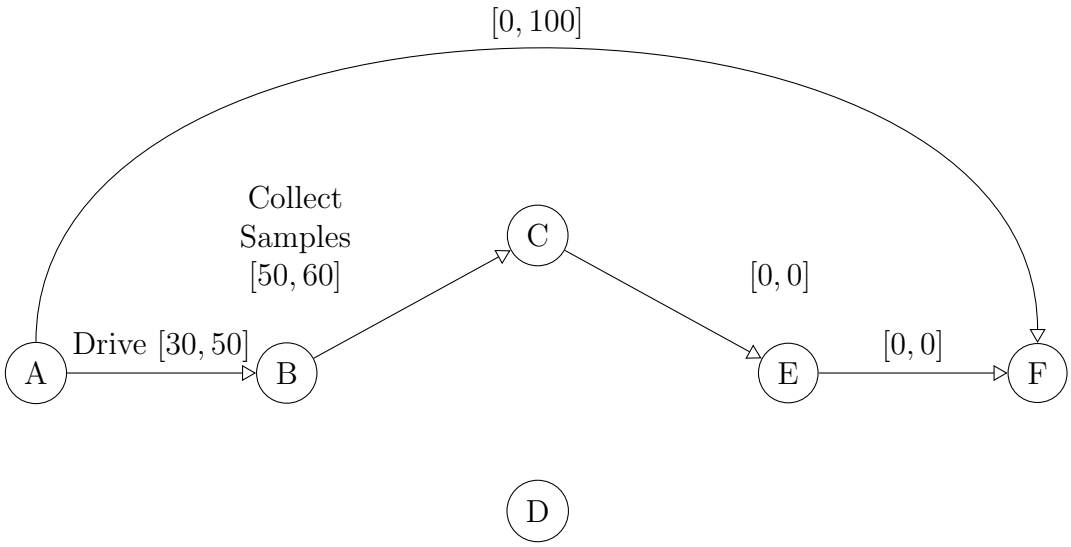
Finally, Drake's use of the ATMS is inspired by Shah's prior work on the compact representation of disjunctive temporal plans for efficient plan execution [12].

The rest of this thesis is organized as follows. Chapter 2 reviews the plan specifications and scheduling frameworks Drake uses and presents the fundamentals of

dispatchable execution that Drake builds upon. Chapter 3 develops the fundamental elements of our compact representation and our compilation algorithm for the deterministic case. Chapter 4 develops the dispatching algorithm for the deterministic case. Chapter 5 applies the ideas behind the deterministic compact representation to extend Drake to handle finite, bounded temporal uncertainty. Finally, we present some performance benchmarks in Chapter 6 and some conclusions and future work in Chapter 7.

Figure 1-1: The component STPs of the rover problem.

(a) Collect Samples Component



(b) Charge Component

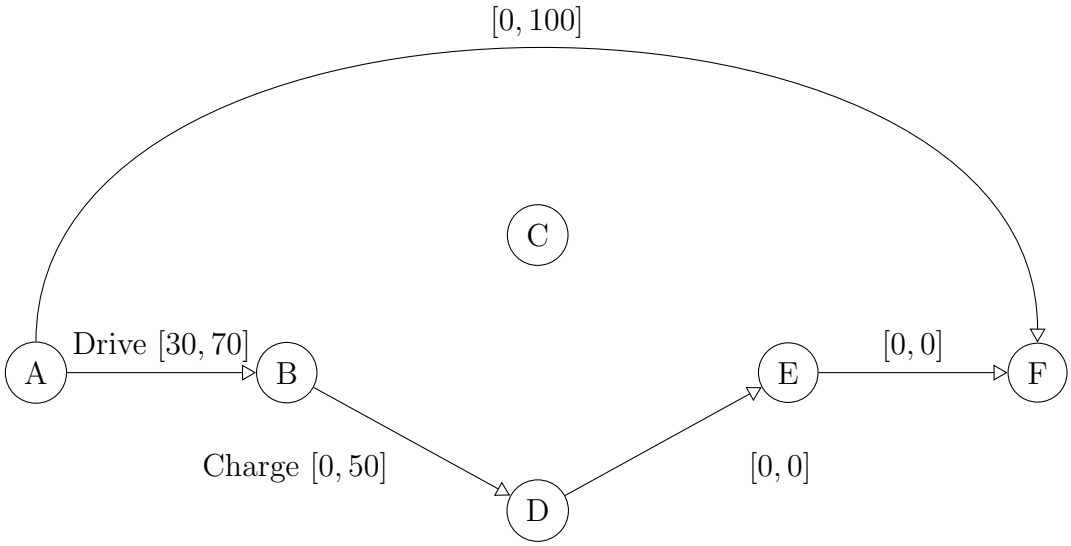
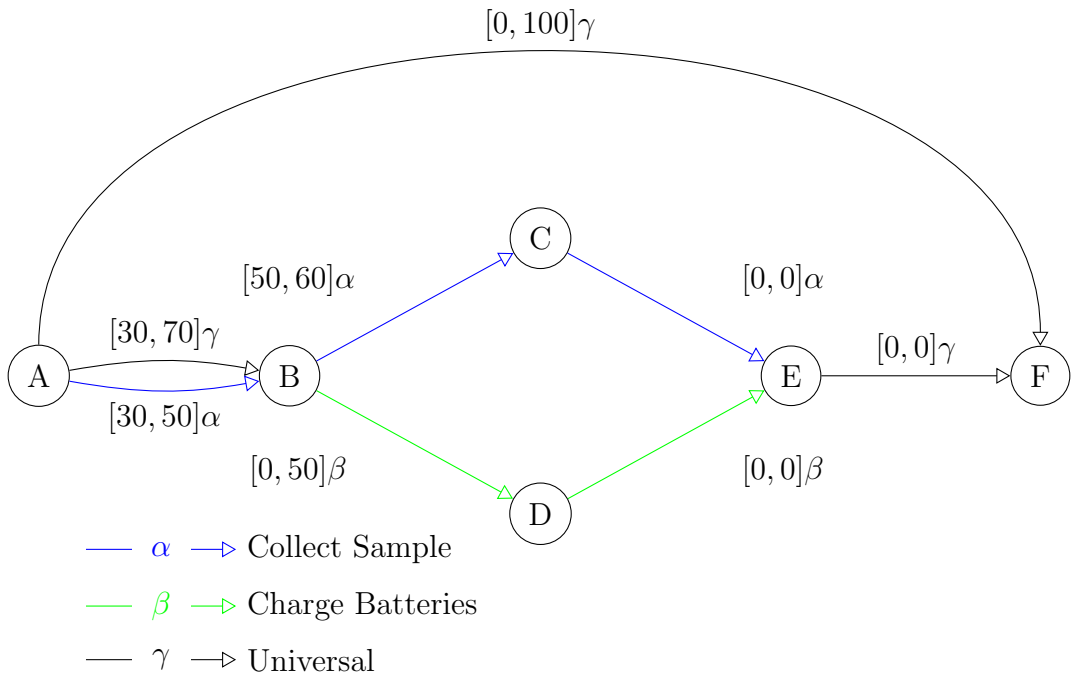


Figure 1-2: A labeled version of component STPs of the rover problem.



Chapter 2

Background and Related Work

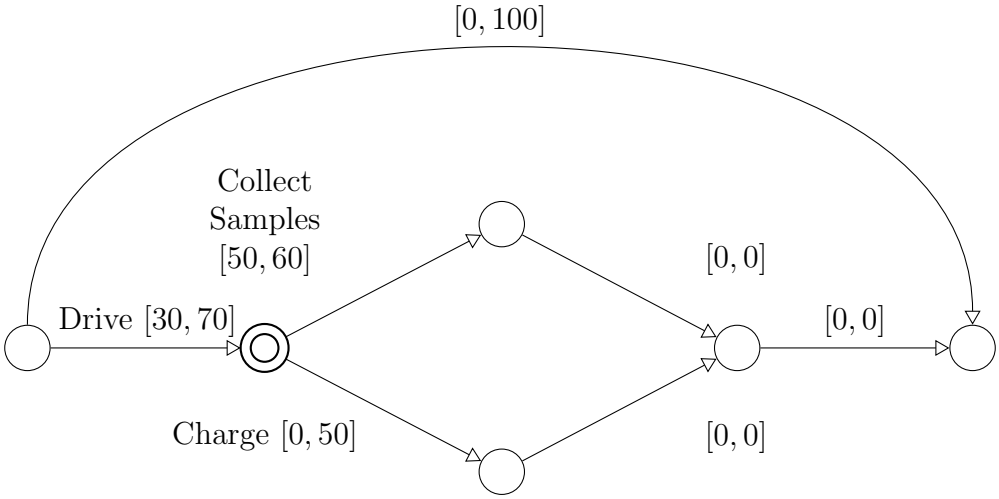
Drake builds upon prior work in plan representation for temporal reasoning and dispatchable execution. While a reader familiar with TPNs, STNs, DTPs, and dynamic execution may wish to skip the following background, it is intended to briefly provide some formal definitions and introduce the reader to the terms used in the literature. Significantly more details of the prior work are given in later chapters, as needed.

First, Section 2.1 defines Temporal Plan Networks, one of the input plan formats employed by Drake. Next, Section 2.2 discusses STPs, the underlying scheduling framework for this work. Section 2.3 introduces Disjunctive Temporal Problems, the other input plan format. Finally, Section 2.4 discusses some motivating prior work into dispatching plans with choice effectively.

2.1 Temporal Plan Networks

To develop the details of delaying decision making and dynamically making choices and schedules, we need a formal way to describe the problems that the executive needs to dispatch. A Temporal Plan Network is a graphical representation for contingent temporal plans introduced by Kim, Williams, and Abramson [8]. The primitive element in a TPN is an activity, comprised of two events connected by a simple interval constraint.

Figure 2-1: This TPN depicts the example from Example 1.1. The rover needs to drive, then collect samples or charge its batteries within a certain time limit.



Definition 2.1 (Event) An instantaneous event in a plan is represented as a real-valued variable, whose value is the execution time of the event. □

Definition 2.2 (Simple Interval Constraint) A simple interval constraint between two events X and Y requires that $l \leq y - x \leq u$, denoted $[l, u]$. □

Simple interval constraints specify that the difference between the execution times of two activities must lie within a particular interval $[l, u]$. Networks are then created by hierarchically composing sub-networks of activities in series, in parallel, or by providing a choice between them. TPNs allow resource constraints in the form of "ask" and "tell" constraints on activities, although Drake does not include algorithms to perform this resource de-confliction. It is also possible, although less common, to place constraint edges between arbitrary nodes in the graph. A TPN therefore provides a rich formalism for expressing plans composed of choices, events, temporal constraints, and activities.

The rover example is depicted as a TPN in graphical form in Figure 2-1. Each of the activities is placed on an arc between the circles, representing events. The double circle node represents a *choice* between outgoing paths, meaning that one set of following activities and events, in the form of a sub-TPN must execute according

to the constraints. The left-most node is the start node and both outgoing arcs denote necessary constraints, representing the drive activity and the overall duration limit. Throughout, the flexible durations are labeled with the $[l, u]$ notation for the lower and upper bound, respectively. The arcs on the right labeled with $[0, 0]$ connect simultaneous events and are present to conform to the hierarchical structure of a TPN.

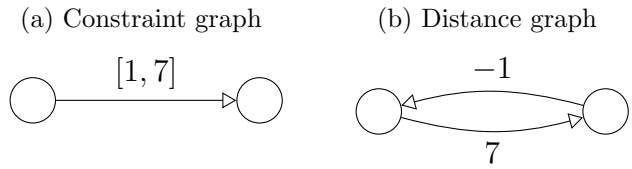
In prior work, Kim et al. presented Kirk, an executive that dynamically makes the scheduling decisions from a TPN [8]. However, Kirk selects a set of choice nodes just before run-time and only makes the scheduling decisions within that choice flexibly. As needed, Kirk triggers a search step to revise the choices. Our work develops a technique for avoiding this tentative commitment to choices until run-time. Before we can describe our dynamic execution strategies, we must develop machinery for reasoning over the temporal constraints, so we now turn to Simple Temporal Problems.

2.2 Simple Temporal Problems

Simple Temporal Networks provide a framework for efficiently reasoning about a limited form of temporal constraint and are the basis of the dynamic execution literature our work builds from. A simple temporal network is defined as a set of events and temporal constraints among them. Events are real-valued time point variables V corresponding to instantaneous events [4]. The time of execution of these events is constrained through a collection of pairwise simple interval constraints as described in Definition 2.2. This constraint is often depicted as $[l, u]$ in Figure 2-2a, where $l \leq u$. By convention, u is non-negative. The lower bound, l may be positive if there is a strict ordering of the events, or negative if there is no strict ordering. Each pair of events is either unconstrained or has exactly one constraint. Positive or negative infinities may be used in the bounds to represent an unconstrained relationship.

A Simple Temporal Network is referred to as a Simple Temporal Problem when the objective is to find a set of real-valued assignments that respects all the constraints of the network. This set of assignments is called a *solution*, and a STP is *consistent*

Figure 2-2: Conversion of a simple STP constraint graph fragment into a distance graph.



if and only if it has at least one solution. Dechter showed that to find a solution or determine consistency, we can reformulate the temporal constraint reasoning as a shortest path problem on a graph [4].

The first step of compiling an STP is to convert the constraints into a weighted graph $\langle E, V \rangle$, where E is the set of weighted edges and V is the set of vertices representing events. First, each event is converted into a node of the graph. Second, the constraints are represented as weighted edges, where each constraint of the form

$$Y - X \leq b_{XY} \tag{2.1}$$

is represented by a weighted edge from X to Y with weight b_{XY}. Intervals specified in $[l, u]$ form, are represented in the distance graph as two edges, one in the forward direction, with weight u and a second in the reverse direction, with weight $-l$. This process is illustrated in Figure 2-2. The distance graph provides a natural expression of the linear constraints because any path length between two events is equivalent to algebraic manipulation on the inequalities. The shortest path between two events specifies the tightest constraint derivable from the algebraic expressions.

Dechter proved that an STP is consistent if and only if its associated distance graph has no negative cycles, which corresponds to an unsatisfiable constraint [4]. This condition can be tested efficiently by computing the Single Source Shortest Path (SSSP) or All-Pairs Shortest Path (APSP) graph. Although the SSSP algorithm is faster, we focus on the Floyd-Warshall APSP algorithm because it is a necessary component of the dispatching process [4, 10]. Note that Dijkstra's Algorithm is unsuitable to compute the APSP graph because the distance graph may have negative

edge weights.

Floyd-Warshall works by stepping through triples of nodes in the graph and looking for shortcuts in the graph. For reference, the algorithm is provided in Algorithm 2.1.

Algorithm 2.1 The Floyd-Warshall All-Pairs Shortest Path algorithm.

```
1:  procedure APSP(V, E)
2:      for i ∈ V do
3:          for j, k ∈ V do
4:              w ← E_ji − E_ik
5:              if w ≤ E_jk then
6:                  E_jk ← w
7:              end if
8:          end for
9:      end for
10:     return E
11: end procedure
```

To prepare the STP for real-time execution, we need to pre-process it so that the dispatcher does not need to perform expensive computations at run-time. Muscettola showed that the STP can be compiled into a *dispatchable form*, where all constraints implicit in the original problem are made explicit, so that the network can be dispatched with only local inference [10]. Since the run-time process only requires local checks, the process is efficient enough to run in real time. Specifically, the run-time algorithm avoids inference over the entire graph.

The APSP form of the distance graph is *dispatchable*, meaning that a dispatcher can make scheduling decisions on-line while only performing propagation to immediate neighbors to guarantee a solution [4]. Since all the constraints implicit in the set C are explicitly enumerated by the APSP graph, the dispatcher can make assignments to the variables without global inference. During execution, the dispatcher tracks *execution windows* that summarize the constraints on each event, which are updated by the local propagations. Some edges imply a strict ordering on which events must be executed first, creating a predecessor and successor relationship between some events; we call these *enablement constraints*. For example, Figure 2-2 depicts ordered events because the right node must occur at least one time unit after the left event. At each

step of dispatching, the dispatcher attempts to finds an event whose predecessors have all been executed and whose execution window includes the current time [10].

Muscettola et al. showed that the APSP contains redundant information, in that groups of edges are guaranteed to propagate the same bound, causing extra work for the dispatcher. Specifically, an edge is said to be *dominated* by another edge, if the dominated edge always propagates the same constraint and therefore is not needed by the dispatcher [10]. These redundant edges may be trimmed after consistency is determined, resulting in a *minimal dispatchable network*. This step reduces the number of edges in the graph and makes run-time processing faster.

2.3 Disjunctive Temporal Problems

A formalism that directly modifies STPs to allow choice is the Disjunctive Temporal Problem (DTP). We use DTPs as a possible problem specification for Drake, because it can provide a family of STPs. Formally, the definition of events are identical to STPs, but each constraint $C_i \in C$ is allowed to be a disjunction

$$c_{i1} \vee c_{i2} \vee ... \vee c_{in}, \tag{2.2}$$

where n may be any positive integer [4]. The disjunctions expand the language of constraints expressible in STNs, allowing new concepts to be expressed, such as non-overlapping intervals. As before, a solution is a set of assignments to each time point in V while meeting at least one simple interval clause of each disjunction in C. The disjunctive constraints make DTPs an expressive formulation, allowing encoding of problems with choices and resources, to name a few important capabilities.

Most modern approaches for determining consistency for DTPs are derived from the observation that a DTP can be viewed as a set of *component* STPs, where the DTP is consistent if and only if at least one of the component STPs is consistent [16, 11, 18]. The component STPs are formed by selecting exactly one simple interval constraint from each disjunctive constraint in the DTP. A solution to any of the

component STPs is a solution of the DTP because it satisfies the simple interval constraints selected from the disjunctions to create the component STP. Therefore, consistency tests can be performed by searching through the possible combinations of disjuncts, in order to find a consistent component STP.

Tsamardinos presented a flexible dispatcher for DTPs that first enumerates all consistent component STPs and then uses them in parallel for decision making [19]. At run-time, the dispatcher propagates timing information in all STPs simultaneously. The dispatcher may make scheduling decisions that violate timing constraints in some of the component STPs, making it impossible to use the corresponding choices, as long as it never invalidates all remaining possible STPs, thus removing all possible choices from the DTP. Drake inherits this strategy for selecting between choices.

2.4 Fast Dynamic Dispatching of TCSPs

Shah et al. approached the problem of dispatching Temporal Constraint Satisfaction Problems, a special case of DTPs, by removing redundant storage and calculations performed by Tsamardinos's algorithm [12, 19]. Temporal Constraint Satisfaction Problems (TCSPs) are DTPs with restricted structure, where for one constraint C_i, every simple interval constraint per DTP constraint involves the same two events, so each constraint can only represent choices between different execution bounds for a given pair of events. It cannot express some DTP concepts, such as non-overlapping constraints. We discuss the details of solving TCSPs further because the insights from their approach inspires Drake's methods.

Shah points out that the component STPs of real-world TCSPs often differ by only a few constraints, hence the space required to store the dispatchable representation can be reduced to keeping only the differences between them. These changes are represented as constraint tightenings on a STP computed by relaxing the original TCSP. These tightenings are computed with an incremental algorithm, called Dynamic Back-Propagation Rules introduced by Stedl and Williams, which only calculate the changes required to maintain dispatchability [15]. Their technique is able

to separate the consequences of certain sets of choices from the overall plan. This technique, although distinct, bears some resemblance to an Assumption Based Truth Maintenance System (ATMS). Shah describes Chaski, an executive that uses these techniques. By avoiding redundant records of shared constraints, Shah's results show dramatic reductions in the size of the dispatchable TCSP [12]. Their dispatching algorithm uses the compact encoding to reduce execution latencies by several orders of magnitude for medium sized problems. Our work is partially inspired by this success, and we explore modifying Chaski's strategy to use a full ATMS labeling scheme, allowing the executive to represent partial assignments to choices. Since DTPs have less common structure than TCSPs, we expect a DTP algorithm would benefit from identifying the shared elements without using a relaxed form of the plan as a base.

The next chapter builds upon the work reviewed in this chapter to develop Drake's compact representation and compilation algorithms. Chapter 4 presents Drake's dispatching algorithms for labeled dispatchable graphs.

Chapter 3

Compilation of Plans with Choice

Recall that Drake is a dynamic executive for Temporal Plan Networks (TPN) or Disjunctive Temporal Problems (DTP), meaning that it selects choices and schedules activities just before execution. Drake does this efficiently by converting either representation into a *labeled distance graph* and then compiling it into a dispatchable form off-line, reducing the reasoning required at run-time. The dispatchable form is computed by applying a variant of the DTP compilation algorithm to the labeled distance graphs. This chapter presents an algorithm that transforms TPNs and DTPs to a compact representation and introduces a novel compilation algorithm that compiles this representation into dispatchable form. The new compilation algorithm adapts Muscettola's STP compilation algorithm to work on a compact representation made possible by a *labeling* scheme, which is inspired by the Assumption-based Truth Maintenance System [10, 2]. The labeling scheme allows Drake to avoid unnecessary repetition in the compiled form of the plans, resulting in a compact form that is more tractable to store and reason over.

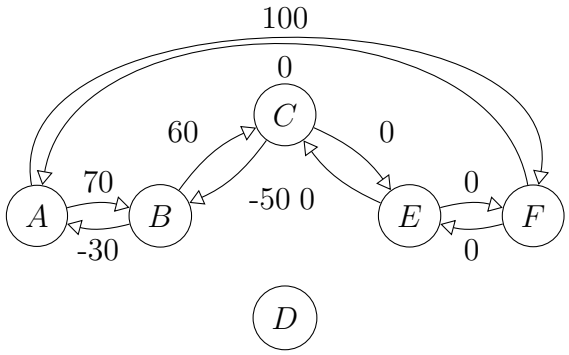
Our novel algorithm is presented in the context of Tsamardinos's compilation algorithm for DTPs [19]. Tsamardinos's algorithm proceeds in three steps: (1) enumerate every component STP, (2) compile each component STP into dispatchable form, and (3) filter each component STP of unnecessary edges. The latter two steps use the standard STP compilation method developed by Muscettola [10]. This prior work introduced the first compiler and dynamic executive for DTPs. All of the essential

steps of the reasoning are directly repeated in our work. Unfortunately, Tsamardinos's compilation procedure is inefficient because each component STP is stored and compiled individually, regardless of any similarities between the STPs that might make some storage or reasoning steps redundant. The enumeration performed in the first step is exponential in the number of choices in the DTP, resulting in a large storage requirement. Drake addresses this problem by exploiting similarities between the component STPs to create an equivalent representation that is substantially more compact.

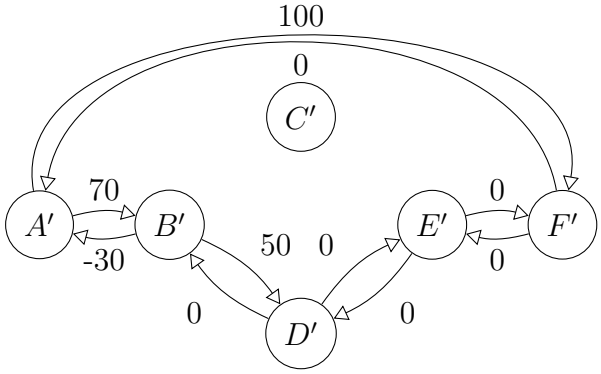
Drake introduces a *labeling* scheme to the constraint storage system, inspired by Assumption-based Truth Maintenance Systems, to efficiently store and perform computations on the constraints [2]. Constraints are annotated with environments, which denote a partial assignment to the choices of the input problem. This partial assignment represents the minimal choices from which the constraint logically follows. Since the STPs have similarities, labeling allows Drake to compactly encode the effects of choices on the constraints. Figure 3-1 shows how Tsamardinos's technique independently records every constraint for every complete combination of choices. In the rover scenario, we can instead record the general deadline constraints only once, and then note separately the implications if the rover decides to collect samples, shown in Figure 3-2. In this figure, we use assignments to the variable x to denote the possible outcomes of the choice. The annotation in curly brackets, for example, $\{x = 1\}$, is an environment specifying that the attached constraint corresponds to a certain assignment to the variables representing the choices. In this case, $\{x = 1\}$ represents collecting samples. We annotate the initial constraints with environments that specify when each constraint holds and then carry the environments through the reasoning steps of compilation. To perform this reasoning, we develop a powerful data structure called *labeled distance graphs*, which stores the values on edges in *labeled value sets*, applying the general framework of labeling and environments provided by the ATMS to take advantage of the structure of the constraints for our application. This formalism allows us to create a compact representation of the temporal constraints and easily modify the existing algorithms to work on the compact representation.

Figure 3-1: The component STN distance graphs of the TPN in Figure 2-1.

(a) Collect Sample Component STN

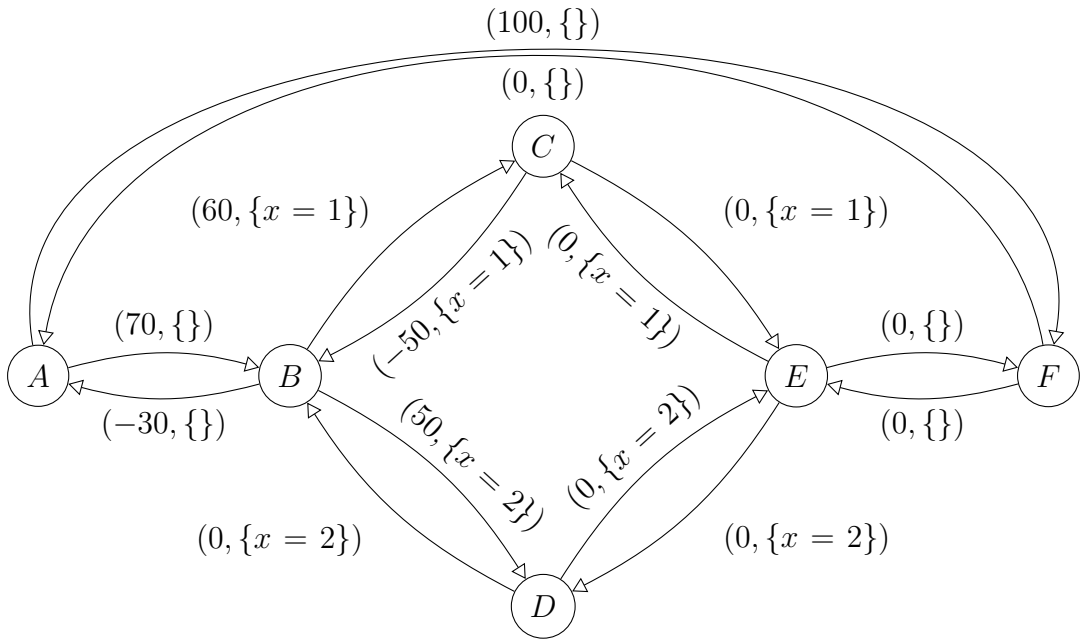


(b) Charge Component STN



To provide a high-level overview of the compilation algorithm we begin with the high-level pseudo-code, presented in Algorithm 3.1. The input to the algorithm is a DTP or TPN, provided as a labeled distance graph described by events V, edges W, and variables X. It either reformulates this graph into a minimal dispatchable form, or signals that the plan is infeasible. A minimal dispatchable form is defined by Muscettola as a form where local propagations are sufficient to perform real-time decision making [10]. First, Line 2 prepares a data structure to hold the conflicts of the labeled distance graph, compactly specifying inconsistent, component STPs. Second, Line 3 compiles the distance graph into dispatchable form with the Labeled-APSP algorithm. This step reveals all the implicit constraints of the problem, recording all the requirements of the input plan explicitly, so that the dispatcher only needs to perform local constraint propagations at run-time. Finally if there are still some

Figure 3-2: The labeled distance graph corresponding to the TPN of Figure 2-1. All edges not drawn are implicitly valued $(\infty, \{\})$. The variable x denotes the choice and has domain $\{1, 2\}$.



consistent component STPs, Line 7 filters the dispatchable graph of unnecessary edges. Some edges are unnecessary because they are guaranteed to propagate the same information at run-time, but the dispatcher only needs to propagate the unique values once, so the redundancy is avoided by removing those edges. This chapter carefully defines both of these phases of the compilation process, which are directly taken from Muscettola [10]. This process creates a compact representation because it avoids the expansion step of Tsamardinos's work, but instead performs algorithms analogous to the STP ones, directly on the compact representation given by the labeled distance graph [19].

This chapter presents the compilation algorithm as follows. First, Section 3.1 motivates the benefits of a labeled encoding with a simple example and provides an overview of some of the necessary formalism. Next, Section 3.2 develops the conversion method from TPNs or DTPs to *labeled distance graphs*, the high-level representation Drake uses to store and reason over the plans, which uses *labeled value*

Algorithm 3.1 Compilation algorithm for Labeled Distance Graphs

```
1: procedure COMPILE(V, W, X)
2:     S ← INITCONFLICTDATABASE(X)
3:     W, S ← LABELED-APSP(V, W, S)
4:     if ENVIRONMENTSREMAIN?(S) then
5:         return null
6:     else
7:         W ← FILTERSTN(V, W)
8:         return W, S
9:     end if
10: end procedure
```

sets to efficiently store the edge weights of the graph that encode constraints. We then develop the formalism for using these techniques. Section 3.3 develops environments to compactly encode the assignments that entail a value. Section 3.4 uses these to create labeled value sets, which Drake uses to store constraints that depend on choices. Finally, we develop the two major steps of the compilation algorithm itself, which use this compact representation. Section 3.5 provides a labeled analog to the APSP algorithm Muscettola uses to compile STPs to dispatchable form. Section 3.6 then removes extra edges the compiler does not need at run-time to create a minimal dispatchable labeled distance graph.

3.1 Introduction to Value Sets and Labeling

Drake's key innovation is the compact representation it uses to represent the family of component STPs. We observe that for real problems, the edge weights of the distance graph encoding each component STP are not unique for each component problem. Rather, the values are loosely coupled to the choices selected for the plan. Therefore, we can avoid the explicit representation of each complete combination of choices explicitly and unify the values for each edge into a single structure called a *value set*. The function of the value set is to losslessly store the values provided and answer queries about the value for that edge for any particular set of choices. Although the value set in general could exactly mimic the expansion Tsamardinos's

work uses, we take the opportunity provided by the unified representation to make that data structure more compact by developing an implementation called a *labeled value set*. Then we represent the entire family of component STPs as a single graph, where each edge has a labeled value set instead of a numeric value. We can easily recreate every component STP by querying all the labeled value sets, but in practice, Drake performs further computations directly on the value sets. By providing generic methods for operating on the labeled value sets, we can easily modify the standard STP compilation routine developed by Muscettola to simultaneously derive the implicit constraints of the problem and the interaction of those constraints with the choices.

The labeled value sets are made efficient by introducing ideas from the Assumption-based Truth Maintenance System (ATMS). That work develops environments, which specify a subset of the choices. Each numeric value is labeled with an environment that specifies the minimal conditions that logically entail the value. Therefore, we can avoid repetitively recording a value that is implied by only a subset of choices. For example, if a particular edge weight has only two unique values and depends on the assignment to only one choice, we can record this with exactly two values, instead of repeating those two values with an exponential number of copies, for every combination of assignments to the other choices. Furthermore, the ATMS provides guidance on how to perform operations with labeled values, directly carrying the choices through the computations that Drake's compiler and dispatcher must perform.

As an introduction to value sets and labeling techniques, we present two examples that motivate their use. First, we identify how Tsamardinos's algorithm performs redundant storage and calculations, show how Drake uses value sets to reduce the size of the representation. Second, we provide a simple linear algebraic system that includes computations similar to those Drake performs, and use labeling to trace the dependencies of the values through the computations.

Example 3.1 Consider the first step of compiling the TPN in Figure 2-1 with Tsamardinos's algorithm. This step creates a distance graph for each component

STP that is implied by a complete set of choices, as shown in Figure 3-3. There are only two component STNs, but in general the number of component STPs is exponential in the number of choices, requiring significant expansion. Some repetitive elements in the two graphs are immediately apparent. For example, the events are all copied between the two graphs and some constraints are identical, such as (A, B) and (A', B'). In fact, only the edges touching events C or D are different. An unusual feature of this graph is that events D and C' are not connected to any events through edges; this means that their execution time is unconstrained.

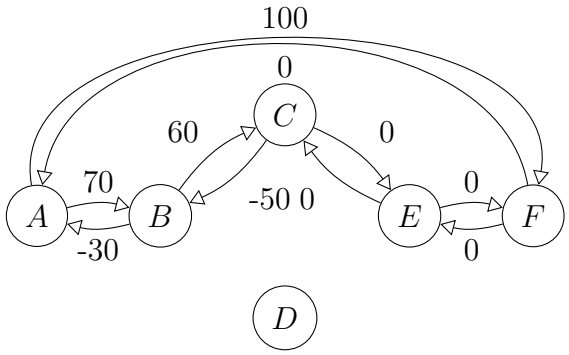
The next step of Tsamardinos's algorithm is to compute the All-Pairs Shortest Path of each graph. Since many of the edges in the two graphs are identical, the APSP will redundantly derive many edge lengths, performing unnecessary work. For example, it will have to independently derive that $w(A, E) = w(A', E') = 100$ twice, once for each component STP, even though every step in the computation is identical. The value of choosing between the two component STPs lies in the weights that change between the two graphs, such as those on (B, C). □

Creating an independent distance graph for each component STP provides a simple mechanism to encode the values of the edges for each complete set of choices, but does so at the cost of redundantly storing and reasoning over any similarities of the graphs. Combining the edge weights of each graph into a single graph with value sets eliminates these redundancies without reducing Drake's ability to encode every implication of the possible choices.

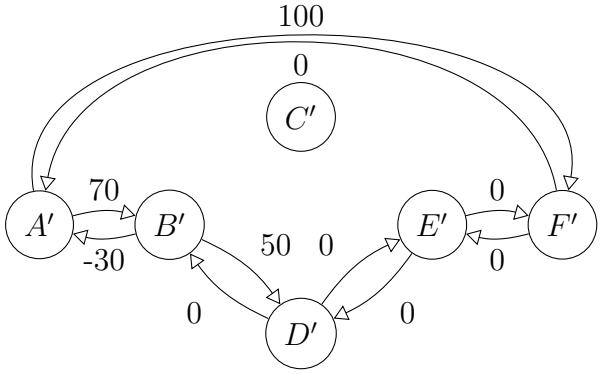
The labeled value set is a general data structure used to record values, where the values are differentiated by their value and the environment the value is labeled with. Our implementation needs to define three basic operations: adding a value, querying for a value, and performing a binary operation on two value sets. During the addition of new values, our implementation maintains minimality of the value set, by pruning the set of any values that the query operator can no longer return. The ability to perform computations directly on the labeled value sets during compilation and dispatch, and then store them minimally is crucial for labeled value sets to form the basis of our compact representation.

Figure 3-3: The component STN distance graphs of the TPN in Figure 2-1.

(a) Collect Sample Component STN



(b) Charge Component STN



Labeled value sets use environments to compactly represent the assignments to choices that logically entail a value, following their use in the Assumption Based Truth Maintenance Systems (ATMS). In the ATMS, de Kleer creates environments to minimally describe the dependence of propositions on assumptions and to derive the dependencies of new propositions [2]. By working with the minimal dependence of the values on choices, we avoid redundancy in both storage and computations. Before rigorously defining the labeling system used by Drake, Example 3.2 illustrates how to use environments to track dependencies through some calculations that are similar to those performed during compilation.

Example 3.2 (Environments and Algebra) Consider a set of real valued vari-

ables, w, x, y, and z, which are related through the equations

$$y = w + 5$$

$$z = y - x$$

If we need to repeatedly compute y and z for various pairs of inputs (w, x) we might consider caching values for later use. For example, if we compute that $(w = 2, x = 1) \rightarrow (y = 7, z = 6)$ we could cache the value for y. However, we need to remember what the value for y depends on, so that we do not incorrectly re-use the value. Here, y depends only on w, so we might record this as "$y = 7$ when $\{w = 2\}$." The sufficient conditions for the value and the assumptions it depends on, are placed in curly brackets and are called the *environment* under which $y = 2$.

Then suppose a query is made of the form $(w = 2, x = a)$. We can immediately see that $y = 7$ because the assignment in the environment is shared with the query, regardless of what particular value a might have. The environment $\{w = 2\}$ is said to *subsume* the inputs because all the assignments match, making the attached fact true. Likewise, if w changes to some novel value, it is clear that we must compute a new value for y.

The variable z has a more complex dependence on the other values and thus has a more complex environment. Any value computed for z is dependent upon both w and x and is labeled with both input values. Hence, the value inherits the *union* of the environments for y and x, leading to labels of the form $\{w = a, x = b\}$. For example, the previous query for $(w = 2, x = 1) \rightarrow (y = 7, z = 6)$ would store the value $z = 6$ with label $\{w = 2, x = 1\}$ because z depends on x directly and on w through its dependence on y. □

In this example the caching has limited utility because we provided a small problem. However, Drake expands this technique to a deep linear system with many layered computations, where separating the dependence saves many redundant operations and reduces the space required to represent all necessary values. The two primary operations Drake performs on environments are *union* and tests for *sub-*

sumption, which we formally define shortly. Note that if every set of choices implies a unique STP, there would be no similarities for this scheme to exploit and the environments would be useless. As we will demonstrate, DTPs display enough common structure for labeling to provide a compact encoding.

3.2 Forming Labeled Distance Graphs

Drake dynamically executes TPNs and DTPs by reasoning about them as families of component STPs, differentiated by a set of discrete choices. Drake is similar to Tsamardinos's work, in that both perform operations on STPs encoded as weighted distance graphs. However, Drake instead uses labeled value sets to efficiently store the edge weights and the variation in the weights caused by the choices of the input problem. We use this new compact encoding scheme to make Tsamardinos's compilation technique more efficient. First, this section presents an encoding for the choices of the input problem and define the *labeled distance graph*, the top level data structure used in Drake's compilation technique. We then describe methods for creating these graphs from TPNs and DTPs. The following section will then build the labeling formalism this representation requires.

Both types of input problems are defined by families of component STPs, where the differences between the components are defined by a set of choices. Drake uses *choice variables* to formally encode the choices.

Definition 3.3 (Choice Variables) Each choice of the input problem is denoted by a finite domain variable x_i. The variable associated with a choice has one domain element d_{ij} for each possible outcome. X is the set of all the variables for a particular problem. □

Example 3.4 In the rover TPN in Figure 2-1, there is one choice. We can replace this with a single variable x. The two choices, collecting samples and charging the batteries, are encoded with the assignments $x = 1$ and $x = 2$, respectively, as seen in Figure 3-2. □

The simple interval constraints of a TPN or DTP have the usual correspondence to a distance graph. We now define labeled distance graphs, which consist of the weights of the distance graph associated with the input plan, and the dependence of the weights on the choices.

Definition 3.5 (Labeled Distance Graph) A labeled, weighted distance graph G is a pair $\langle V, W \rangle$. V is a list of vertices and W is a group of labeled value sets (see Definition 3.14) with domination function $f(a, a') := (a < a')$ (see Definition 3.13), representing the weight function that maps the ordered vertex pair and an environment (see Definition 3.7) to weights: $V \times V \times \mathcal{E} \rightarrow \mathbb{R}$, for any vertex pair $(i, j) \in V \times V$ and environment $e \in \mathcal{E}$. The set of edges E contains those pairs where $w(i, j) \neq \infty, w \in W$, for some environment. All the labeled value sets in W are initialized with the pair $(\infty, \{\})$. □

This definition uses *labeled value sets*, which are an efficient storage mechanism for values that vary with different assignments to the choice variables. Differences in the choices are recorded in *environments*, summarizing the set of assignments to the choice variable that entail some consequence. These are formally defined in the next sections.

The distance graph needs to store only the shortest known path between vertices, hence the domination function f is the less-than inequality. STP reasoning in general requires shortest paths because each path represents an inequality between two time points, and only the tightest bound is important. Again, we define domination functions for labeled value sets carefully in the next section, but the key idea is that the labeled value set exploits the importance of small values to improve efficiency. Note the choice of the strict inequality; the labeled value set must store exceptions to the existing values and selecting $f(a, a') \leftarrow (a \leq a')$ needlessly creates apparent exceptions.

Building upon this definition, we now explain how to create a labeled distance graph representing a TPN. Recall that a TPN consists of a set of events, constraints, and choices. Algorithm 3.2 converts a TPN into a labeled distance graph. First, Line

2 creates the nodes of the graph to be the same as the nodes of the TPN. Second, Line 3 creates one choice per choice node, and a value for each corresponding to the outgoing arcs from those choice nodes. Finally, Lines 4-6 encode each simple interval constraint into the labeled weight function. As usual, each temporal constraint $[l, u]$, where l and u are real numbers, produces one forward edge with weight u and one backward edge with weight $-l$.

The difference between this algorithm and the process of converting a STP into its distance graph is that here we add environments to each edge. The next section will define environments formally, but an environment specifies a partial set of choices from which the constraint logically follows. Setting the environments correctly in the initial formulation ensures that the following algorithms derive the correct dependencies of constraints on the choices. Remembering that TPNs are generally hierarchical, the environment for an edge must specify the assignment for all choices that occur higher in the hierarchy. This technique ensures that the entire sub-plans of a choice are mutually exclusive, as required by a TPN.

Algorithm 3.2 Convert a TPN into a labeled distance graph

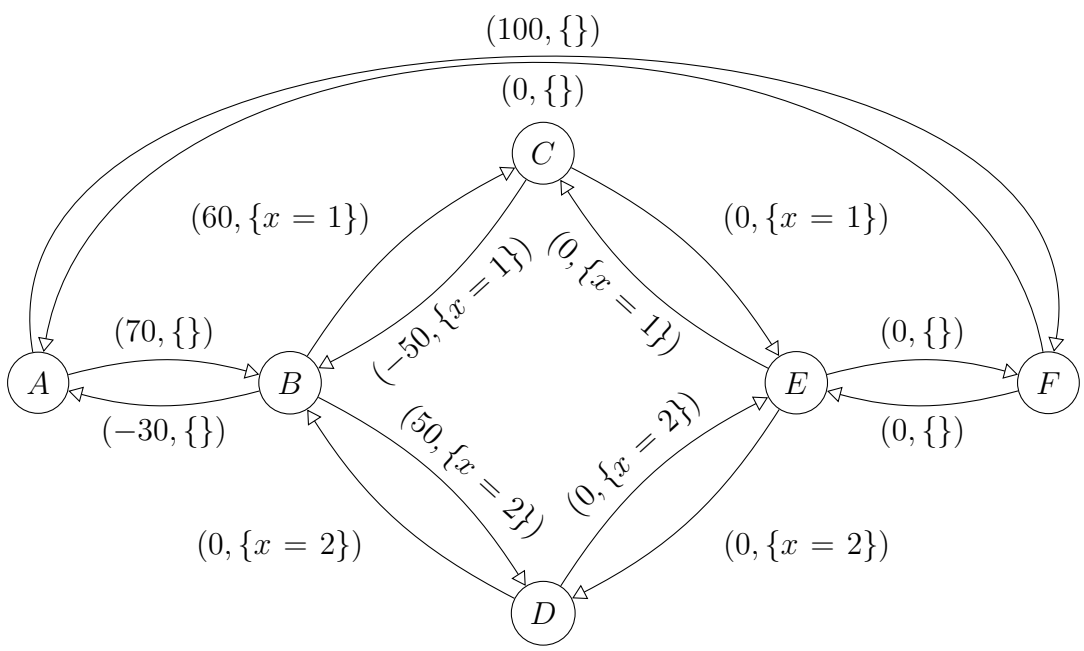
```
1: procedure TPNToDGraph(TPN)
2:     V ← events of TPN
3:     X ← CreateChoiceVariables(TPN)
4:     for each temporal constraint in TPN do
5:         add weights to W, labeled with all choices higher in the hierarchy
6:     end for
7:     return V, W, S, X
8: end procedure
```

Continuing our example from the rover TPN, we demonstrate running TPNToD-Graph on it.

Example 3.6 The first step of compiling the TPN in Figure 2-1 is to transform it into a labeled distance graph. The final result is shown in Figure 3-4. The choice node requires a variable, denoted x, with possible values $x = 1$ corresponding to collecting samples, and value $x = 2$ for charging. The constraints are transformed into distance graph edges in the usual way: upper bounds are positive distances in the forward

direction and lower bounds are negated on backward edges. However, edges along the path of choices are labeled with the environment for the choice that activates those edges. Therefore, edges (B, C) and (C, E) are labeled with $x = 1$, and edges (B, D) and (D, E) are labeled with $x = 2$. The other edges are not conditioned on any choices and are given an empty label to indicate that they represent constraints that must hold regardless of the which option is selected. Both of the component distance graphs in Figure 3-3 can be recovered from this compact form, yet there are no duplicated constraints, as seen in Figure 3-3. □

Figure 3-4: The labeled distance graph corresponding to the TPN of Figure 2-1. All edges not drawn are implicitly valued $(\infty, \{\})$. The variable x denotes the choice and has domain $\{1, 2\}$.



Converting a DTP into a labeled distance graph requires a nearly identical process. The choices from a DTP are created from the disjunctive clauses. A DTP uses inclusive-or operators, indicating that the executive needs to enforce at least one disjunct for the execution to be correct. We can accommodate by creating a variable for each disjunctive clause and one value for that variable for each disjunct. For example, if some DTP has a disjunction

$$(A - B \leq 5) \vee (A - B \geq 3) \vee (A - C \leq 3) \tag{3.1}$$

then we could create a single variable x for this choice, with a domain $\{1, 2, 3\}$, for each disjunct, respectively. The choice variable notation implies an exclusive choice, implying that it must commit to a single disjunct, which is not implied by a DTP. This strategy is correct, however, because the choice selects the single disjunct, Drake satisfies to ensure correctness, without prohibiting any of the other disjuncts from being satisfied incidentally. For example, if $x = 1$, then Drake commits to satisfying the first constraint, which does not generally prohibit Drake from satisfying the third constraint. If the executive needed to explicitly reason over satisfying multiple disjuncts from a disjunctive clause, we could instead create a value for each combination of satisfied clauses. For example, we could create domain values $\{4, 5, 6\}$ to explicitly consider satisfying pairs of disjuncts, although Drake does not do this.

With the variables defined, the constraints from DTPs are simply converted into the labeled distance graph. Non-disjunctive constraints are labeled with empty environments, specifying that they must always be satisfied. Each disjunctive clause is labeled with an environment specifying the single variable and value that corresponds to that disjunct.

This section gave an algorithm to convert TPNs or DTPs into labeled distance graphs. This step mirrors the first part of Tsamardinos's compilation algorithm: to create distance graphs for all the component STPs. Our compact representation allows Drake to efficiently reason over the choices and temporal constraints. With this high level goal in mind, the next section provides a full development of labeled value sets.

3.3 Environments and Conflicts

This section provides the formal definition of an environment and defines the essential operations performed on them. Drake uses environments to minimally specify the choices that imply a constraint and to derive the dependence of implicit constraints on

the choices. The *subsumption* operation is used to determine if a particular environment contains the assignments specified by another, which Drake uses to determine if a value is compatible with a certain scenario. The *union* operator is necessary for computations involving labeled values. The definitions in this section exactly follow de Kleer's work and are necessary background to develop the efficient implementation of labeled value sets described in Section 3.4 [2].

An environment is an assignment to a subset of the choice variables that summarizes the sufficient conditions for some derivation or computation to hold. The following definition is loosely based on de Kleer's notation [2]. Drake builds environments with assignments to choice variables. We do not allow an environment to assign a multiple distinct values to any variable, in order to maintain logical consistency.

Definition 3.7 (Environment) An *environment* is a list of assignments to a subset of the variables in X, written $\{x_i = d_{ij}, ...\}$. There may be at most one assignment to each variable; hence, an environment of the form $\{x_i = d_{ij}, x_i = d_{ij'}, ...\}$ is forbidden. A *complete environment* provides exactly one assignment to each variable in X. An empty environment provides no assignments and is written $\{\}$. We denote the set of possible environments as $\mathcal{E}$ and the set of complete labels as $\mathcal{E}_c$. The length of an environment is the number of assigned variables, denoted $|e|$. □

Example 3.8 Given one variable x with a domain of two values $(1, 2)$, the set of possible environments is $\mathcal{E} = \{\{\}, \{x = 1\}, \{x = 2\}\}$. This set includes the empty environment, which makes no assignments, and every possible partial and full assignment to all of the variables. In this case, there are no partial assignments and only two full assignments. The set of complete environments is both non-empty environments $\mathcal{E}_{\rceil} = \{\{x = 1\}, \{x = 2\}\}$. □

In the ATMS, a proposition may be labeled by a set of environments, where each environment logically entails that proposition [2]. Drake gives each proposition exactly one environment, but the proposition may occur multiple times. This design decision is made for ease of implementation, and while sufficient for our purposes, is not required. In fact, de Kleer's ATMS maintains unique propositions because they

simplify some ATMS operations and may provide performance benefits, and therefore, reintroducing them to our work is an avenue for future research.

Subsumption and *union* are the fundamental operations Drake performs on environments. Subsumption is used to determine if one environment provides the requirements of another. Union is the primary way to combine environments to create new ones.

Definition 3.9 (Subsumption of Environments) An environment e subsumes e' if for every assignment $\{x_i = d_{ij}\} \in e$, the same assignment exists in e', denoted $\{x_i = d_{ij}\} \in e'$. □

This definition only specifies the inclusion of the assignments from e in e', but does not prohibit other assignments to e' for variables which are unassigned in e.

Example 3.10 Given variables x_1, x_2 with domains $(1, 2)$, then:

- $\{x_1 = 1\}$ subsumes $\{x_1 = 1, x_2 = 2\}$, because the only requirement is that the subsumed environment has the assignment $x_1 = 1$ from the first environment.
- $\{x_1 = 1\}$ subsumes $\{x_1 = 1\}$, because the assignments are exactly the same.
- $\{\}$ subsumes $\{x_1 = 1\}$, because the first provides no assignments and therefore there are no requirements on the subsumed environment.
- $\{x_1 = 1, x_2 = 1\}$ does not subsume $\{x_1 = 1\}$, because one of the assignments of the subsuming label $x_2 = 1$ is not present.
- $\{x_1 = 1, x_2 = 1\}$ does not subsume $\{x_1 = 1, x_2 = 2\}$, because the labels disagree on the assignment to x_2. □

From the definition of subsumption, a few useful properties arise. First, all environments subsume themselves. Second, every environment is subsumed by an empty environment, because an empty environment imposes no assignments. Finally, a complete environment subsumes only itself because every distinct environment of the

same length must differ in one or more assignments, and any shorter environment must not assign one of the variables assigned in the complete environment.

The union operation is used when performing an operation on labeled values, because the union creates a new environment that contains the assignments of two environments.

Definition 3.11 (Union of Environments) The *union* of environments, denoted $e \cup e'$ is the union of all the assignments of both environments. If e and e' assign different values to some variable x_i, then there is no valid intersection and the empty set, $\emptyset$, is returned instead. The empty set signifies that there is no environment where both e and e' hold simultaneously. □

Example 3.12 Let there be variables x_1, x_2 with domains $(1, 2)$.

- $\{x_1 = 1\} \cup \{x_2 = 2\} = \{x_1 = 1, x_2 = 2\}$
- $\{x_1 = 1\} \cup \{x_1 = 1\} = \{x_1 = 1\}$
- $\{x_1 = 1\} \cup \{\} = \{x_1 = 1\}$
- $\{x_1 = 1\} \cup \{x_1 = 1, x_2 = 2\} = \{x_1 = 1, x_2 = 2\}$. Intersecting an environment with one that subsumes it always produces the subsumed environment.
- $\{x_1 = 2\} \cup \{x_1 = 1, x_2 = 2\} = \emptyset$, because they disagree on the assignment to x_1. □

An important function of an ATMS is the ability to track inconsistent environments. In our case, an inconsistent environment signals an inconsistent component STP. Drake must keep track of choices that are inconsistent with one another and which component STPs are still possible. The standard strategy in an ATMS is to keep a list of minimal *conflicts* or *no-goods* [2]. Tsamardinos's approach maintained a list of the valid component STPs. The conflict strategy, which uses environments to keep a minimal summary of the inconsistent environments, is much more compact.

A *conflict* is an environment that states the minimal conditions necessary to entail an inconsistency [22]. For example, the compilation process might determine

that $x_1 = 1$ and $x_2 = 1$ are contradictory, such that they cannot both be selected, regardless of any other assignments to other choices. Then, $\{x_1 = 1, x_2 = 1\}$ is a conflict of the system. All environments subsumed by this conflict also contain the inconsistency and are invalid. For example, $\{x_1 = 1, x_2 = 1, x_3 = 1\}$ is inconsistent because it is subsumed by the conflict. All environments not subsumed by the conflict do not contain the inconsistency and are not known to be invalid. Therefore, conflicts are used to summarize the inconsistent environments.

In contrast, *kernels* summarize the environments that are currently known to be valid. An environment subsumed by a kernel, by definition, is not subsumed by any conflicts and therefore is consistent [22]. A kernel must differ from each conflict by at least one assignment to ensure that no environment is subsumed by both a conflict and a kernel. The minimal kernels of the conflict $\{x_1 = 1, x_2 = 1\}$ are $\{x_1 = 2\}$ and $\{x_2 = 2\}$, assuming that both variables have domains $\{1, 2\}$, because they represent the two ways of avoiding the conflict. In this example, $\{x_1 = 2, x_3 = 1\}$ is consistent because it is subsumed by the kernel $\{x_1 = 2\}$.

The other important function of the conflict database is to determine if all the complete environments have been invalidated, or of a certain conflict would do so. For example, assume there is a variable $x_1 \in \{1, 2\}$. If both $\{x_1 = 1\}$ and $\{x_1 = 2\}$ are conflicts, then regardless of any other variables in the problem, there are no complete assignments possible, because neither possible assignment for a variable is feasible. Therefore, the entire problem is inconsistent. During compilation and dispatch, Drake must keep track of whether there are choices available so it can avoid failure where possible, and signal the failure if necessary.

Our algorithms use a database of conflicts and kernels to determine if an environment is known to be valid. In our pseudo-code, we call the conflict database data structure S. This database keeps a set of minimal conflicts and updates the kernels to be consistent with the conflicts. The database is used throughout the algorithms to test for environment consistency. We now define the functions Drake uses to interact with the conflict database, but leave details of the algorithms required to Williams et al. [22].

The two fundamental operations in Drake are testing an environment for consistency and adding conflicts. EnvironmentValid?(S, e) returns false only if the environment e is subsumed by a conflict and is inconsistent. Otherwise it returns true. AddConflicts(S, e) adds the environment e as a conflict. It returns true if there are no remaining consistent complete environments, otherwise it returns false.

The function EnvironmentsRemain?(S) returns a Boolean value, and returns true if any consistent complete environments remain. This is easily testable because it is true exactly when there is at least one kernel of the system. ConflictsPossible?(S, l) queries the database of whether making conflicts from the environments in list l would invalidate all the complete environments. For example, when there are no conflicts, $\{x_1 = 1\}$ can be a conflict without removing all the options because $\{x_1 = 2\}$ is still available. Then later, $\{x_1 = 2\}$ is not a possible conflict because, as described before, that would remove all the possible complete environments. This function would then allow Drake to avoid creating that conflict. The function takes a list of environments because the dispatcher may need to test whether it is allowed to add multiple conflicts simultaneously.

CommitToEnv(S, e) modifies the conflict database to ensure that all the remaining consistent, complete environments are subsumed by e. To accomplish this, the function creates conflicts as necessary. For example, if there were no conflicts and the dispatcher took an action that required $\{x_1 = 1\}$, then it would commit to that environment and remove any contradictory options from consideration. In this example, this requires creating a conflict for $\{x_1 = 2\}$.

The function InitConflictDatabase(X) simply initializes a new conflict database to have no conflicts for variable descriptions X.

Environments provide a technique for Drake to succinctly state the dependence of a proposition on the choices of the plan. The subsumption and union operations are used when determining if a labeled value is appropriate and during computations on labeled values, respectively. Similarly, the conflict database provides an efficient mechanism to determine which component STPs are consistent and to reason about steps that might invalidate them. With this formalism defined, we can explain the

implementation of labeled value sets.

3.4 Labeled Value Sets

This section describes in detail the implementation of labeled value sets. We describe this implementation carefully before moving on to the compilation algorithm because the compact representation is the core contribution of this work. This also allows us to explain where the details of the implementation impact the higher level algorithms. The purpose of the labeled value sets is to allow Drake to compactly map from choices in the input problem to constraints implied by those choices. This section provides a formal definition and gives algorithms for manipulating the labeled value sets. We also prove the correctness of the algorithms. With the tools from this section, the derivation of the compilation algorithm in the remaining sections proceeds naturally.

Section 3.2 informally explained that the labeled value sets only need to keep the tightest constraints, which are given by the smallest edge weights. Each edge weight represents an inequality, where for some pair of events A and B and an edge weight l, $B - A \leq l$. If there are two bounds l and l', where $l < l'$, then l' specifies a looser constraint and is not needed. This feature of handling inequalities in an ATMS is developed by Goldstone, who *hibernates* propositions that are unnecessary, keeping them from redundantly entering into computations [7]. We use the same idea to prune weaker inequalities, when permitted by the environments associated with the inequalities.

Labeled distance graphs only need the less-than inequality, but at dispatch, execution windows also require labeled value sets with the greater-than inequality to keep the tightest lower bound, which is the largest value. Therefore, we define a general *domination function* that specifies the inequality. We say that a tighter value *dominates* a looser value.

Definition 3.13 (Domination Function) The domination function $f(a, a')$ provides a total ordering over all possible values of a, returning true if a dominates a'. $f(a, a)$ returns false. For any pair of distinct values a and a', either $f(a, a')$ or $f(a', a)$

must return true. □

Drake only uses strict inequalities for the domination function, which clearly provide a total ordering over all real numbers. We specify that calling the domination function with identical arguments should return false because this would allow identical values to appear different.

Now we can present the definition of a labeled value set. Intuitively, it is a list of values that are labeled with environments.

Definition 3.14 (Labeled Value Set) A *labeled value set* for domination function $f(a, a')$ is a set A of pairs (a, e) where a is a value and $e \in \mathcal{E}$ is an environment. □

Drake interacts with labeled value sets through three operations: query, add, and binary operations. The first two operations read and store to the value set, respectively. Binary operations are used to derive new labeled value sets from existing ones.

The query operator is designed to find the dominating value that is appropriate for some environment. A value might be returned if its environment subsumes the input environment. Of the possible values, the dominant value is returned. Formally:

Definition 3.15 (Labeled Value Set Query) The query operator $A(e)$ returns a_i from the pair $(a_i, e_i) \in A$ where e_i subsumes e and $f(a_i, a_j) = true$, for all e_j present in any pair of A where e_j subsumes e. If no environment e_j subsumes e, then $A(e)$ returns $\emptyset$. □

Adding to the labeled value sets simply requires placing the new labeled value into the set.

Definition 3.16 (Adding to Labeled Value Sets) Adding the labeled value (a, e), with value a and environment e to the value set, requires updating the labeled value set $A \leftarrow A \cup (a, e)$. □

At this stage, the labeled value sets may not be compact because the set might contain redundant values. The following example illustrates how the structure of

domination and subsumption can help prune the value set. We then use this structure to design an algorithm to add values to the labeled value set that also maintains the minimality of each set.

Example 3.17 Again, consider variables x_1, x_2 with domains $(1, 2)$ where A uses

$$f(a, a') := a < a'$$

and is initialized to $A = \{(5, \{\})\}$. A call to $A(e)$ for any environment $e \in \mathcal{E}$ produces five because every environment is subsumed by the empty environment. Then suppose add to the value set that $x_1 = 1$ is a sufficient condition for the value to be three. Adding the value produces

$$A = \{(3, \{x_1 = 1\}), (5, \{\})\}.$$

Any query environment that contains $x_1 = 1$ is subsumed by both environments in the labeled value set, making both values possible candidates. However, three is dominant over five, and is therefore returned. Querying the labeled value set for other environments matches the empty environment and returns five. Now imagine that we add the labeled value $(2, \{x_1 = 1\})$. Similarly, the new pair is added to the set.

$$A = \{(2, \{x_1 = 1\}), (3, \{x_1 = 1\}), (5, \{\})\}$$

Notice that $A(e)$ does not return three for any input environment e, because any e subsumed by the environment of three is also subsumed by the two's identical environment, and two dominates three. The value A can be accurately represented with only the two and five terms, consequently saving space and search time for queries. □

Now we give a theorem stating that this form answers queries with no loss of information and prove the correctness of the uniqueness criteria.

Theorem 3.18 (Minimality of Value Sets) *A valued label set may be pruned of all subsumed non-dominant values, leaving a minimal set, without changing the result of any possible query $A(e)$.* □

PROOF For a labeled value set A, assume for contradiction that there is some pair (a_i, e_i) that fails the uniqueness criteria, but cannot be discarded because it is required to correctly answer the query $A(e)$. If it fails the uniqueness criteria then there is another pair (a_j, e_j) where e_j subsumes e_i and $f(a_j, a_i) = \mathrm{T}$. The i pair can only influence the query if it provides the correct returned value. If a_i is the proper returned value, then by definition, e_i subsumes e. However, e_j must also subsume e because subsumption is transitive, as is easily demonstrated by considering the assignments implied by subsumption. Then, both a_i and a_j are candidates responses, and we would select the dominant value, a_j. Since a_i would not be selected for any environment e, it could have been discarded, which contradicts the assumption. ■

Algorithm 3.3 provides an incremental update rule for adding values to labeled value sets, maintaining a minimal representation by removing all values that cannot be returned by any query, as motivated by Example 3.17. The input to the function is an existing labeled value set A, the new labeled value set B, which may be non-minimal, and the domination function f for A. The output is the labeled value set, updated with the new labeled value if it is useful and can be returned, and any values the new value supersedes are pruned. The outer loop simply processes each value of the new value set B.

To illustrate this algorithm, reconsider the last step of Example 3.17. In that example, the labeled value set is $\{(3, \{x_1 = 1\}), (5, \{\})\}$ and we need to add the value $(2, \{x_1 = 1\})$. The algorithm proceeds in two steps. First, Lines 3 - 7 search through the existing set and make sure that the new value's environment is not subsumed by the environment of any dominant values. If so, the new value is not needed and the algorithm returns without modifying A. The environment of the new value, $\{x_1 = 1\}$ is subsumed by the identical environment in the labeled value set, but the value 3 is not dominant over 2, so this condition is not triggered and the value is useful, and

should be added to the set.

If the value needs to be kept, Lines 7 - 12 find and remove any pairs whose environments are subsumed by the new value's environment and dominated by the new value. In this example, the new labeled value subsumes the environment and dominates the value $(3, \{x_1 = 1\})$ from the labeled value set because the environments are identical, and $2 < 3$. Therefore, this value is removed from the labeled value set after the new value is added. The labeled value $(5, \{\})$ remains because $\{x_1 = 1\}$ does not subsume $\{\}$. Finally, Line 13 adds the new value to the possibly reduced set, and returns, producing the expected result given in Example 3.17.

Drake uses this function whenever values are added to labeled value sets to maintain the compactness of their representation.

Algorithm 3.3 Add new elements to a labeled value set, maintaining minimality.

```
 1: procedure AddCandidates(A, B, f)                ▷ Add labeled values in B to A
 2:     for (b_i, e_i) ∈ B do                                   ▷ Loop over new values
 3:         for (a_j, e_j) ∈ A do                        ▷ Test if new value is needed
 4:             if (e_j subsumes e_i)&&(f(a_j, a_i) == T) then
 5:                 continue A                   ▷ Not needed, continue to next value
 6:             end if
 7:         end for

 8:         for (a_j, e_j) ∈ A do                                ▷ Check all old values
 9:             if (e_i subsumes e_j)&&(f(a_i, a_j) == T) then
10:                 A ← A \ (a_j, e_j)                   ▷ Old value redundant, prune
11:             end if
12:         end for
13:         A ← A ∪ (a_i, e_i)
14:     end for
15:     return A
16: end procedure
```

The final step in developing the formalism for these structures as an efficient value storage technique is to define operations on them. We require a way to calculate $C = g(A, B)$ for arbitrary functions g where A, B, and C are all labeled value sets. During compilation, Drake uses the value sets to store edge weights and needs to compute $C = A + B$. However, we develop this operation generally because Chapter 5 uses it to apply several different propagation rules. First, we show a technique for

performing operations on individual pairs of values with environments. This technique is directly inspired from de Kleer's work [2].

Theorem 3.19 (Operations on Values with Environments) *For some pair of labeled values (a, e_a) and (b, e_b) from the labeled value sets A and B, any deterministic function of two inputs g produces a labeled pair $(g(a, b), e_a \cup e_b)$.* □

PROOF If e_a is an environment for a, meaning that e_a entails a, and likewise e_b is an environment for b, then any deterministic function of a and b is entailed by the union of all the assignments in e_a and e_b. ■

The union of the input environments may produce an empty set if the environments for a and b have contradictory assignments, indicating that the value $g(a, b)$ is never logically consistent. For example,

$$(4, \{x = 1\}) + (2, \{y = 1\}) = (6, \{x = 1, y = 1\})$$

In this case, the new value of 6 is labeled with a new environment that requires the assignments to both x and y given by the input environments. In contrast,

$$(4, \{x = 1\}) + (2, \{x = 2\}) = (6, \emptyset)$$

The environment is not possible because the two input environments assign the variable x to have different values. Therefore, this value is never appropriate and the pair should be discarded. In general, the new value results from applying the function g to the input values and that the new environment is always the union of the two input environments.

Applying binary operations to entire labeled value sets requires taking the cross product of the input sets. This is justified by Theorem 3.20.

Theorem 3.20 (Binary Operations on Labeled Values) *For two labeled value sets A and B containing complete representations of their respective values, a set*

$C = g(A, B)$ for some deterministic function g is defined by the set of candidate values $(g(a_i, b_j), e_{ai} \cup e_{bj})$ for all i, j. □

PROOF Since the list of values a_i and b_j are the only possible values under any environment, the output of a deterministic function must come from the evaluation of the cross product of those lists. As given in Theorem 3.19, the union of their environments is the environment for each new value. Alternatively, the definition of the correct values for C is

$$C(e) = g(A(e), B(e))$$

where for an input environment e we query for the correct values of A and B, then compute function g. To pre-compute the result for all environments, setting $e = e_{ai} \cup e_{bj}$ puts the least possible requirements on e while being certain that the input values are entailed by the environment of the result. ■

We need one further step to derive an algorithm for performing binary operations on labeled value sets; Theorem 3.19 specifies the new labeled values that define the new set, but does not specify how to create a minimal representation of C that considers the domination function for C. However, this is precisely what ADDCANDIDATES is designed to do. We compute the candidate values by applying g to the cross product of the values of the two input sets and incrementally add the candidate values to a new set. Algorithm 3.4 implements this technique for labeled value sets, computing all the terms of the cross product with a double loop and adding each value with a valid environment into a minimal set. Line 2 initializes C with an empty value set.

To illustrate this algorithm, consider performing the operation

$$\{(5, \{\}), (3, \{x = 1\})\} + \{(6, \{\}), (5, \{x = 1\}), (3, \{y = 1\}), (2, \{x = 2\})\}$$

where the domination function is $f :=<$ and $\{x = 1, y = 1\}$ is a conflict of the system. Lines 3-11 loop over all pairs of values from A and B to create the cross

product of values. The actual candidate value provided by the function g and the union of the environments are computed on Lines 5.-6. In this case, the candidates are:

$$(11, \{\}), (10, \{x = 1\}), (9, \{x = 1\}), (8, \{x = 1\}), (7, \{x = 2\}), (8, \{y = 1\}), (5, \emptyset), (6, \{x = 1, y = 1\})$$

The next step is to ensure that we only add values to the new set if the environment is not known to be invalid. In this example, the value $(6, \{x = 1, y = 1\})$ has an environment that subsumes the conflict of the system, so this value is discarded. Additionally, non-existent environments are also removed, so $(5, \emptyset)$ is removed at this stage. Line 7 checks both these conditions before Line 8 updates C. Finally, the candidates are added to the new labeled value set, and during this process $(10, \{x = 1\})$ and $(9, \{x = 1\})$ are discarded because they is not needed in a minimal representation of a labeled value set because of the value $(8, \{x = 1\})$. Therefore, the final result is:

$$C = \{(11, \{\}), (8, \{x = 2\}), (7, \{x = 2\}), (8, \{y = 1\})\}$$

Algorithm 3.4 Calculate the results of a binary operation on a minimal dominant labeled value set.

```
procedure LabeledBinaryOp(A, B, f, g, S)                ▷ Compute C ← g(A, B)
    C ← {}
    for (a_i, e_i) ∈ A do
        for (b_j, e_j) ∈ B do
            c_ij = g(a_i, b_j)                          ▷ Calculate the candidate
            e_ij = e_i ∪ e_j
            if (e_ij ≠ ∅) ∧ EnvironmentValid?(e_ij) then    ▷ Keep if valid
                C ←AddCandidates(C, {(c_ij, e_ij)}, f)
            end if
        end for
    end for
    return C
end procedure
```

This section has defined labeled value sets, a compact representation for values

that depend on assignments to discrete values. These structures also use the ordering of the value, formalized as domination, to represent the labeled value sets with a minimal set of labeled values. We also defined how to add to value sets, query them for values, and perform mathematical operations on them, while maintaining the minimality of the representation. With these definitions, we have defined all the necessary tools to derive Drake's compilation algorithm. Labeled value sets are a powerful representation for ordered values, which we use to store the edge weights in labeled distance graphs and other temporal bounds used during compilation and dispatch. This data structure relies heavily on prior work in ATMS, and underlies the compact representation that provides the performance improvements of this thesis [2, 7]. Furthermore, the operations we have defined here allow us to simply integrate this data structure into existing algorithms. The following section uses these functions to define a variant of the All-Pairs Shortest Path algorithm for labeled distance graph, which we use as the first phase of the compilation algorithm.

3.5 Labeled All-Pairs Shortest Path

Next, we consider how to compute the dispatchable form of labeled distance graph. Recall that Muscettola proved that compiling to dispatchable form involves making all the temporal relationships between the events explicit. Then we trim redundant constraints where, at compile-time, we can remove some edges without affecting the deductions made by the dispatcher. The constraints are exposed by applying a variant of the Floyd-Warshall All-Pairs Shortest Path algorithm, which is developed in this section [1]. The labeled distance graph is a compact representation of all the initial component STPs, so a single run of this new algorithm compactly computes all the compiled component STPs in a single step, replacing Tsamardinos's need to compile them individually. The standard Floyd-Warshall algorithm is almost sufficient to perform these computations; we only modify it to interact with the labeled value sets using the operators developed in Section 3.4. Section 3.6 describes the edge filtering process.

One important aspect of Tsamardinos's technique is that some of the component STPs may be marked invalid if negative cycles are found by the APSP algorithm, because a negative cycle implies an inconsistency in the constraints such that the STP has no solution. Drake identifies these inconsistencies on the fly and creates conflicts for them. This allows Drake to terminate immediately if all the solutions are removed and avoid computations only relevant to known inconsistent component STPs. Avoiding computations for inconsistent environments is a standard technique for improving efficiency in an ATMS [2]. Recall that the plan is dispatchable if at least one component STP is dispatchable.

Algorithm 3.5 Labeled APSP Algorithm

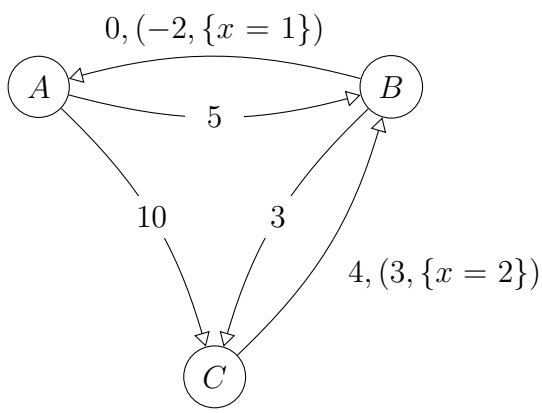
```
 1: procedure LABELED-APSP(V, W, S)
 2:     for i ∈ V do                                   ▷ Cycle through triangles
 3:         for j, k ∈ V do
 4:             C_jk ← W_ji + W_ik                     ▷ Apply "+" with 3.4
 5:             if j == k then                         ▷ Self-loop update
 6:                 S ← CHECKFORNEGCYCLES(C_jk, S)
 7:             else                                   ▷ Non-self-loop update
 8:                 W_jk ← ADDCANDIDATES(W_jk, C_jk, '<')   ▷ Alg. 3.3
 9:             end if
10:         end for
11:     end for
12:     return W, S
13: end procedure

14: procedure CHECKFORNEGCYCLES(C_jk, S)
15:     for (a_i, e_i) ∈ W_jk where a_i < 0 do         ▷ for all negative cycles
16:         S ← ADDCONFLICT(S, e_i)    ▷ find inconsistent environments, Sec. 3.3
17:         if ENVIRONMENTSREMAIN?(S) then             ▷ Sec. 3.3
18:             signal inconsistent DTP
19:         end if
20:         REMOVEFROMALLENV(e_i)
21:     end for
22:     return S
23: end procedure
```

The Labeled All-Pairs Shortest Path Algorithm is based on the Floyd-Warshall and is shown in Algorithm 3.5. Recall that the Floyd-Warshall algorithm updates the shortest paths by looking for a route $j \rightarrow i \rightarrow k$ that provides a smaller weight

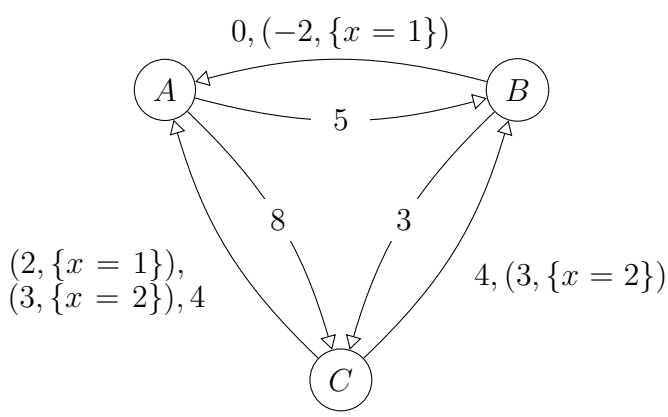
than the weight on the existing edge $j \to k$. The base LABELED-APSP algorithm is nearly identical to the standard Floyd-Warshall algorithm [1]. There are three differences. First, the addition required to derive new path lengths is computed on labeled value sets, so the labeled value set operation is used. Second, self-loops are not stored, but are checked for negative cycles to find inconsistent component STPs. Third, the non-self-loop candidates are added to existing labeled value sets with the ADDCANDIDATES operation.

Figure 3-5: A simple example of running Labeled-APSP. Unlabeled values have an implicit empty environment. For example, 10 represents $(10, \{\})$

(a) Input labeled distance graph



(b) Output APSP labeled distance graph



We can illustrate this algorithm by compiling the small distance graph in Figure 3-5 with a single choice $x \in \{1, 2\}$. Stepping through each step of the Floyd-Warshall algorithm is tedious for even three nodes, so we present selected steps. The outer

for-loops iterate through triangles of the graph, deriving shorter path lengths. Line 4 computes the path lengths that two sides of the triangle imply for the third with the labeled binary operation function given by LabeledBinaryOp, instead of the scalar operation. Line 6 checks for negative cycles when creating self-loops to detect inconsistencies. Finally, Line 8 stores any derived values not on self-loops, by updating the old labeled value set with the newly derived values.

In our example, consider the non-self-loop update steps. Only the labeled value sets on edges (A, C) and (C, A) are revised. First, $w(A, C)$ is revised with $w(A, B) + w(B, C)$. The only candidate pair is $(8, \{\})$, which has a shorter path than the existing value $(10, \{\})$, while having the same environment, so the old value is replaced. Now $w(C, A)$ is revised with $w(C, B) + w(B, A)$. Each of those weights has two labeled values, leading to the candidate values in the following table

Source (w_{CB}, l_{CB})	Source (w_{BA}, l_{BA})	Candidate (w_{CA}, l_{CA})
$(4, \{\})$	$(0, \{\})$	$(4, \{\})$
$(3, \{x = 2\})$	$(0, \{\})$	$(3, \{x = 2\})$
$(4, \{\})$	$(-2, \{x = 1\})$	$(2, \{x = 1\})$
$(3, \{x = 2\})$	$(-2, \{x = 1\})$	$(1, \emptyset)$

The first line shows the derivation of a 4 with an empty environment, where the empty environment is inherited from both the inputs. The second and third line show the propagation of a labeled value through a value with an empty environment, producing a labeled value with the sum of the values and the same non-empty environment. The final line does not receive an environment because the two environments give competing values for x and their union is therefore the empty set. The remaining three pairs are first stored in C_{jk} and are then merged into the labeled value set for $w(C, A)$. Note that the value of 4 in the table is not strictly necessary, because the component STPs actually assign x to have some value. Since both values of x have dominant entries in the table, no actual component STP uses the value of 4. Therefore, $(4, \{\})$ is not necessary in a minimal representation, but our algorithms do not identify this, because this conclusion requires reasoning about more than two

labeled values simultaneously. We do not repair this shortcoming, but we give some further discussion of this issue in Chapter 7.

No further propagations update any of the labeled value sets and the updated graph is shown in Figure 3-5. To illustrate the self-loop update, consider computing the self-loops for C created by following the path to B. This path induces $(7, \{\})$ and $(6, \{x = 2\})$ self-loop candidates for C. Since neither one is negative, there are no inconsistencies found by CheckForNegCycles. If there was a negative edge weight, Line 16 would make a conflict for it, and then return failure if all the environments are inconsistent. If there remain consistent environments, signaling that some component STPs may be dispatchable, then Line 20 calls RemoveFromAllEnv, which we do not provide pseudo-code for, but which removes from every labeled value set, every labeled value whose environment is subsumed by the new conflict to avoid storing information about inconsistent STPs.

This variant of the Floyd-Warshall algorithm does not have polynomial run-time because the number of pairs in the labeled value sets is not polynomially bounded. Instead, the worse case bound is the number of component STPs of the input plan, multiplied by the $\mathrm{O}\big(N^3\big)$ of Floyd Warshall.

Now we can prove the correctness of this algorithm. The proof is somewhat clearer if we consider running the APSP algorithm first, then performing the post-processing as first suggested. However, the version presented is more efficient because it avoids performing any propagations with values for inconsistent component STPs.

Theorem 3.21 *The* Labeled-APSP *function shown in Algorithm 3.5 produces a labeled representation of the APSP of all the consistent component STPs of the input DTP.* □

Proof The computation of the APSP form of the graph depends only upon the correctness of the Floyd-Warshall algorithm and on the operations of labeled value sets. The requirement to derive the shortest paths by definition means that all edge weights are dominant with $f(a, a') \leftarrow a < a'$. The only operation required on labeled sets is addition, which is proved correct by Theorem 3.19. All the invalid component STPs

are identified and discarded by negative self-loop edge weights, as in the unlabeled case. ■

Having completed our presentation of the labeled APSP algorithm, it is instructive to re-interpret Tsamardinos's algorithm within the new terminology. Tsamardinos's technique separates the representation of the STP for each complete environment, removing any need to explicitly represent the environments. The benefit of handling the environments in the new method, however, is that each calculation done with a partial environment derives the same information as repeating that propagation in all the component STPs whose complete environments are subsumed by the incomplete environment. In general, this can lead to exponential savings in the number of computations, where the exponent is the number of unassigned variables in the partial environment. Therefore, we can think of each propagation performed by the labeled algorithm on a partial environment to be equivalent to a batch of operations across the component STPs.

The idea of batching leads to a characterization of the performance of the labeled algorithm. If there are N choices of order d, then Tsamardinos's algorithm immediately creates d^N component STPs individually. Instead, the labeled algorithm only splits the environments as necessary to represent distinct values. In the worst case, the labeled algorithm might derive a different weight for each complete environment for every edge, leading to the same storage requirement as Tsamardinos's work. However, as we show empirically later, most real problems include limited interactions between the constraints because the component STPs are often related and not tightly constrained. Instead, the number of distinct environments in the graph, and therefore the number of values in the labeled value sets, is only exponential in the length of the longest label in the compiled graph, $\mathrm{O}\big(d^{\max|e|}\big)$. If the choices do not interact to create unique component STPs, the environments might be short, leading to a small number of total values in each edge's labeled value sets. Dechter quantifies a similar effect in general constraint satisfaction problems by computing the *tree-width* of a constraint problem, representing the true complexity of the interactions of the constraints. While constraint problems are easily solved in time and

space exponential in the total number of variables, advanced and/or search graph algorithms can be reduced in both time and space complexity to be exponential in only the tree-width [3]. Compiling a plan for dispatch is simply constraint propagations, so it seems reasonable that a more complex algorithm than Tsamardinos's explicit expansion, like Drake's, can see similar reduction in the exponent.

This section used labeled value sets to derive an efficient algorithm to compute the All-Pairs Shortest Path of a graph with weights that depend on the selection of a set of choices from a TPN. We also showed the simplicity of modifying the standard Floyd-Warshall algorithm to work with the compact representation. We also gave some theoretical evidence for its performance benefits over prior work.

3.6 Pruning the Labeled Distance Graph

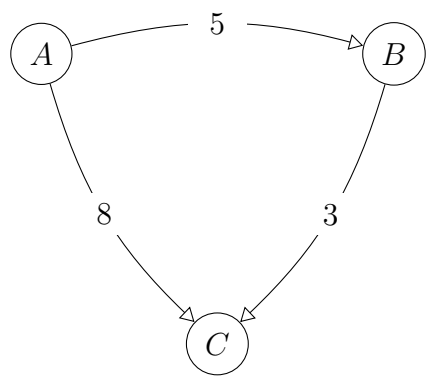
Muscettola et al. developed a post-processing step for dispatchable networks to prune redundant edges [10]. Although the APSP form of the graph is dispatchable, at run-time, many edges are guaranteed to re-propagate the same values in a way that can be identified at compile time. Pruning these edges can drastically reduce the space needed to store the solution and the number of propagations necessary at run-time, without affecting the correctness of the dispatcher. This section develops a direct extension of this useful technique for the labeled graphs [10].

The following theorem, reproduced from Muscettola's work, identifies the edges that may be removed. Essentially, whenever there is a triangle in the graph that has the same weights on one edge as on the sum of the other two edges, then one of the edges is not necessary.

Example 3.22 For example, Figure 3-6 shows a small graph fragment. In this graph, edge (A, C) is dominated by edge (B, C), because the same bound is propagated. Specifically, say A is executed at time $t = 0$. Then, B must be executed before $t = 5$ and C must be executed before $t = 8$. Whenever B is executed, the outgoing edge produces a constraint that C happens within three time units, which ensures that C executes before time $t = 8$. The bound of 8 cannot be broken without first executing

B, and a tighter bound is created through the edge (B, C). Therefore, the edge (A, C) is not needed for correct run-time deductions. □

Figure 3-6: An example of edge domination. Edge (A, C) is dominated by the other edges of the triangle.



Generally, we can summarize domination with the following theorem:

Theorem 3.23 (Edge Domination[10]) *Consider a consistent STN where the associated distance graph satisfies the triangle inequality.*

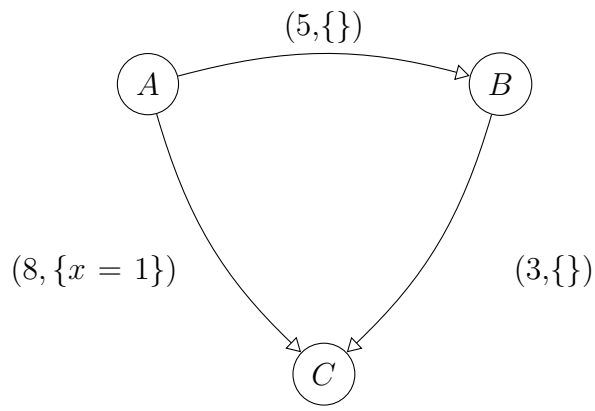
1. *A non-negative edge (A, C) is upper-dominated by another non-negative edge (B, C) if and only if $w(A, B) + w(B, C) = w(A, C)$.*

2. *A negative edge (A, C) is lower-dominated by another negative edge (A, B) if and only if $w(A, B) + w(B, C) = w(A, C)$.* □

This theorem is summarized by Figure 3-8. Intuitively, upper domination holds if the weight propagated through positive weight path $A \to C$ is exactly propagated through some other path $A \to B \to C$ and if the propagation is guaranteed to create the bound in time for the executive to enforce it. The non-negativity ensures that the propagation from B to C actually happens before it is needed. Lower domination is the inverse. The APSP form of the dispatchable graph always satisfies the triangle rule, so the domination test is a sufficient condition to allow edge pruning. The only necessary extension Drake requires is to derive the equivalent rule in the labeled case, as demonstrated in Example 3.24.

Example 3.24 If we add environments to the previous example, as shown in Figure 3-7, the domination is unchanged. The propagation through the 5 and 3 edges creates a bound with an empty environment, which is at least as tight as the one created by the 8. Since the 8 bound has an environment subsumed by the union of the other two value's environments, we know that the domination holds in all the component STPs where (A, C) is entailed. □

Figure 3-7: An example of labeled edge domination. Edge (A, C) is dominated by the other edges of the triangle.

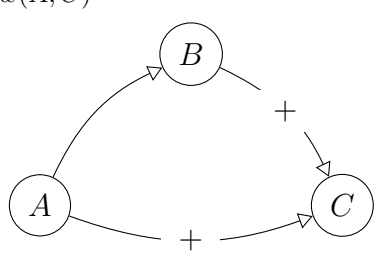


We generalize this to a new theorem.

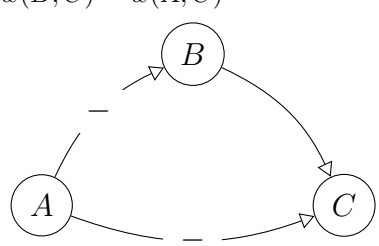
Theorem 3.25 (Labeled Edge Domination) *Consider a consistent labeled distance graph that satisfies the triangle inequality. Consider a triangle of edge weights,* $(w_i^{AB}, e_i^{AB}) \in W(A, B), (w_j^{AC}, e_j^{AC}) \in W(A, C)$*, and* $(w_k^{BC}, e_k^{BC}) \in W(B, C)$*.*

Figure 3-8: Schematic of edge pruning rules for STPs.

(a) *Upper Domination.* Non-negative edge AC is upper-dominated by non-negative edge BC if $w(A, B) + w(B, C) = w(A, C)$



(b) *Lower Domination.* Negative edge AC is lower-dominated by edge weight AB if $w(A, B) + w(B, C) = w(A, C)$



1. *A non-negative, non-zero edge weight* w_i^{AC} *is upper-dominated by another positive, non-zero edge weight* w_k^{BC} *if and only if* $w_i^{AB} + w_k^{BC} = w_j^{AC}$ *and* $(e_i^{AB} \cup e_k^{BC})$ subsumes e_j^{AC}

2. *A negative, non-zero edge weight* w_i^{AC} *is upper-dominated by another negative, non-zero edge weight* w_k^{AB} *if and only if* $w_i^{AB} + w_k^{BC} = w_j^{AC}$ *and* $(e_i^{AB} \cup e_k^{BC})$ subsumes e_j^{AC} □

Proof The edge is dominated if the triangle equality is exactly met under all necessary environments. Since we seek to dominate edge (A, C), in both cases we need the environment of both sides of the equality to hold in at least all the edges of the value used for (A, C). As shown previously, the environment of the sum of two labeled values is given by the union of their environments. Subsumption tests whether this union holds for all the necessary labels. ■

We add the non-zero clause to the domination requirement to ensure that our dispatcher is guaranteed to obey the ordering for the constraint propagations.

Except for the alteration of the domination theorem, the filtering algorithm is identical to the one given in Muscettola's work, presented in Algorithm 3.6 for completeness. The algorithm searches for dominated edges. A subtle point is that the entire searching process must complete before removing any edges. Edges are not deleted immediately because one edge may be dominated by another edge that is scheduled for removal, and maximal filtering does not occur if the edges are removed immediately after they are identified.

The algorithm proceeds as follows. Line 2 searches over all edges with a shared vertex. If the two edges are mutually dominant, then Lines 3-6 arbitrarily mark one for deletion, unless one was marked previously, in which case the marking is left unchanged. Otherwise, Lines 7-9 mark the single dominated edge for deletion. Finally, Line 11 removes all the marked edges.

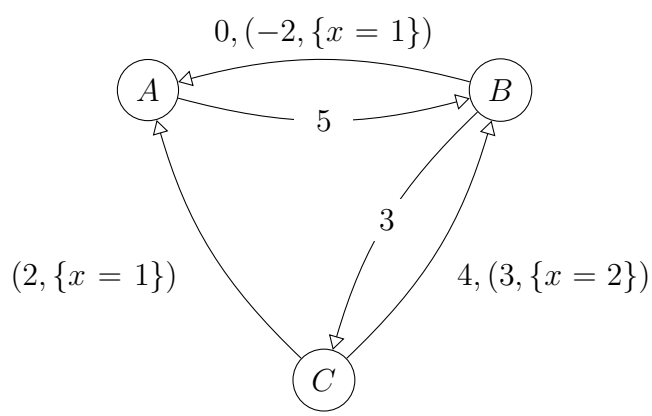
Example 3.26 We illustrate the filtering algorithm by continuing from the APSP labeled distance graph shown in Figure 3-5. Again, because of space considerations

Algorithm 3.6 Filter the compiled Labeled APSP graph of unnecessary edges [10].

```
procedure FilterSTN(V, W)
    for each pair of intersecting edges weights in G do
        if both dominate each other then
            if neither is marked then
                arbitrarily mark one for elimination
            end if
        else if one dominates the other then
            mark dominated edge for elimination
        end if
    end for
    remove all marked edge weights from graph
    return W
end procedure
```

we only identify the dominated edges. First, $W(C, A) = (8, \{\})$ is dominated by the path $C \to B \to A$ because the weights are the same and all the environments are empty. Likewise, the 4 on (C, B) and the 0 on (B, A) dominate the 4 on (C, A). Considering the labeled edges, the weight 3 on (C, A) is dominated by the 0 on (B, A) and the 3 on (C, B) because the weights satisfy the triangle inequality and $(\{\} \cup \{x = 2\})$ subsumes $\{x = 2\}$. Each of these edges is not mutually dominated, so each is marked during the search process and then deleted at the end. As in this case, it is common for many of the derived weights of the labeled APSP graph to be removed through this filtering process, resulting in Figure 3-9. □

Figure 3-9: The filtered DTP from Example 3.26.



To summarize, Muscettola's filtering algorithm is a post-processing step for dis-

patchable networks, designed to reduce the space required to store the dispatchable graph and the number of propagations required during dispatching [10]. This section adapts that algorithm to work on labeled distance graphs. Note that there is another algorithm for STPs, which interleaves the APSP computation and edge filtering, avoiding the expansion and contraction that is characteristic of the APSP and filtering process [17]. This other algorithm could likely be modified with our labeling technique, and we briefly discuss the possibility in Chapter 7. We did not base our implementation on this algorithm because it requires a more complex graph algorithm, but that algorithm has a lower run-time. This filtering process completes our development of a compilation algorithm for labeled distance graphs.

3.7 Summary of Compilation Algorithm

This chapter has developed an efficient technique for compiling TPNs or DTPs into a dispatchable form. This method uses a labeling system to provide a compact representation of the dependence of the temporal constraints of the problem on the choices in the input problem. We developed a formalism for the compact representation, labeled distance graphs and labeled value sets, from prior work on the ATMS [2]. Using these data structures, we developed a compilation algorithm that required relatively simple modifications to the standard STP dispatcher developed by Muscettola, yet is equivalent in expressivity to the explicit enumeration Tsamardinos performs [10, 19].

Chapter 4

Dispatching Plans with Choice

This chapter presents Drake's dispatching technique for DTPs and TPNs, where the input plan is represented as a family of possible STPs, differentiated by discrete choices. Drake is able to make the discrete choices dynamically, such as selecting between collecting samples and charging in our rover example. Drake does this by following Tsamardinos's strategy of executing the possible compiled, component STPs in parallel, making it possible for the executive to consider the possible alternatives [19]. Drake is differentiated by its use of labeled distance graph formalism developed in Chapter 3 to compactly represent the compiled form of the component STPs.

In this chapter, we describe how to take the original STP dispatcher developed by Muscettola and update it to work with labeled value sets and labeled distance graphs [10], yielding Drake's dispatching algorithm for deterministic problems. This dispatcher uses labeled value sets to compactly encode the temporal bounds required for dispatching, in addition to using the labeled distance graph representation of the dispatchable form. This chapter also modifies Muscettola's STP dispatcher to schedule *activities*, which are components of TPNs used to represent some real-world processes [8]. Activities require a minor departure from the STP formalism, but are more realistic in some cases because they specify that durations might need to be scheduled ahead of time, rather than after the duration is already complete.

Recall that Drake interprets its input plan as a collection of component STPs and is responsible for scheduling the events of the plan, such that all the constraints of at

least one component STP are satisfied. We wish to delay the selection of choices so the dispatcher is more robust to disturbances at run-time, which requires a more complex procedure. Tsamardinos approached this problem by maintaining every compiled STP in memory and by updating them all in parallel, allowing the executive to immediately determine which choices any particular scheduling decision is consistent with. Thereby, the executive can begin the execution with all its options available, and incrementally select between them as the execution unfolds. Crucially, it can perform this selection without any risk of unknowingly removing all the remaining choices. Tsamardinos's changes to Muscettola's dispatcher maintain the property that the dispatcher only needs to perform local reasoning for run-time decision making. Drake adopts this broad strategy, implemented with labeled distance graphs.

The objective of this chapter is to present Drake's dispatching algorithm for deterministic problems. This algorithm takes as input a labeled distance graph, which has been compiled and found dispatchable as described in Chapter 3. The output of this algorithm is an execution of the problem, where each event is scheduled dynamically, such that all the constraints of one complete set of choices are satisfied. Put another way, at the end of the execution, Drake must select a single component STP from the labeled distance graph, where it has satisfied every constraint in that component STP. Drake should select this component STP dynamically, avoiding committing to choices earlier than necessary. Finally, the dispatcher restricts itself to local reasoning steps to keep dispatching tractable, as both Muscettola's and Tsamardinos's dispatchers do.

We present Drake's dispatching algorithms as follows. First, Section 4.1 reviews Muscettola's dispatcher and provides the top-level algorithm for Drake. Then, Section 4.2 discusses the techniques Drake uses to compute and store execution windows, a fundamental operation for STP dispatchers. Section 4.3 presents the algorithms for selecting which events to execute, providing the second principal part of the dispatcher. Section 4.4 describes test for identifying missed execution windows, which is necessary for flexible execution. Next, Section 4.5 discusses the extensions required to handle activities. Finally, Section 4.6 concludes.

4.1 Dispatching Overview

Muscettola proved that an STP dispatcher can guarantee the correct execution of a compiled STP through a greedy reasoning process that only performs one-step propagation of temporal bounds [10]. This top level algorithm is quite simple. Essentially, it loops, searching for events to execute, until either every event is executed or a failure is detected. Determining if an event is executable only requires determining if the constraints between the event and its neighbors are satisfied, which is performed in two steps: testing that the quantitative relationships encoded in the distance graph are satisfied and testing that the ordering constraints are satisfied.

Recall that in a STP encoded as a weighted distance graph, each edge represents an inequality between the execution time of two events, specifying the quantitative relationships in the problem. The compilation process reveals implicit constraints that the dispatcher needs to obey, also which are discovered and explicitly encoded as edges. For example, if there is an edge from event A to B with a weight of five, then that edge represents the inequality

$$B - A \leq 5. \tag{4.1}$$

If, for example, event A is executed at time $t = 3$, then the dispatcher can substitute in that execution time and rearrange the inequality

$$B - 3 \leq 5 \tag{4.2}$$

$$B \leq 8 \tag{4.3}$$

The result of the rearrangement is an upper bound on the execution time of event B, created by propagating the execution time of event A one step through the graph. Propagating through an edge in the other direction yields a lower bound, for example, if B executed before A. In general, the dispatcher performs this propagation efficiently by incrementally revising *execution windows* for each event. Execution windows are

the tightest upper and lower bounds found for each event through the propagation of execution times. Checking that an event is within its execution window is sufficient to ensure the quantitative constraints are met, if the ordering constraints are also satisfied. This incremental strategy allows us to propagate the constraints once, rather than testing all the constraints at each time step an event is tested for executability.

Testing whether the ordering constraints of an event are satisfied is called testing for *enablement*. A simple temporal constraint between two events may imply a strict ordering or may be unordered. For example, a constraint $[5, 10]$ between two events A and B specifies that B must happen between five and ten time units after A. This implies ordering, so there is no possibility that A could legally occur before B. On the other hand, if we place a constraint $[-10, 10]$ between events C and D, this specifies that the two events must occur within ten time units of each other, but does not care whether C or D happens first. The dispatcher must test enablement of an event independently from the execution windows to ensure that all the propagations that must update the execution windows actually occur before the execution window is used to justify the execution time of an event.

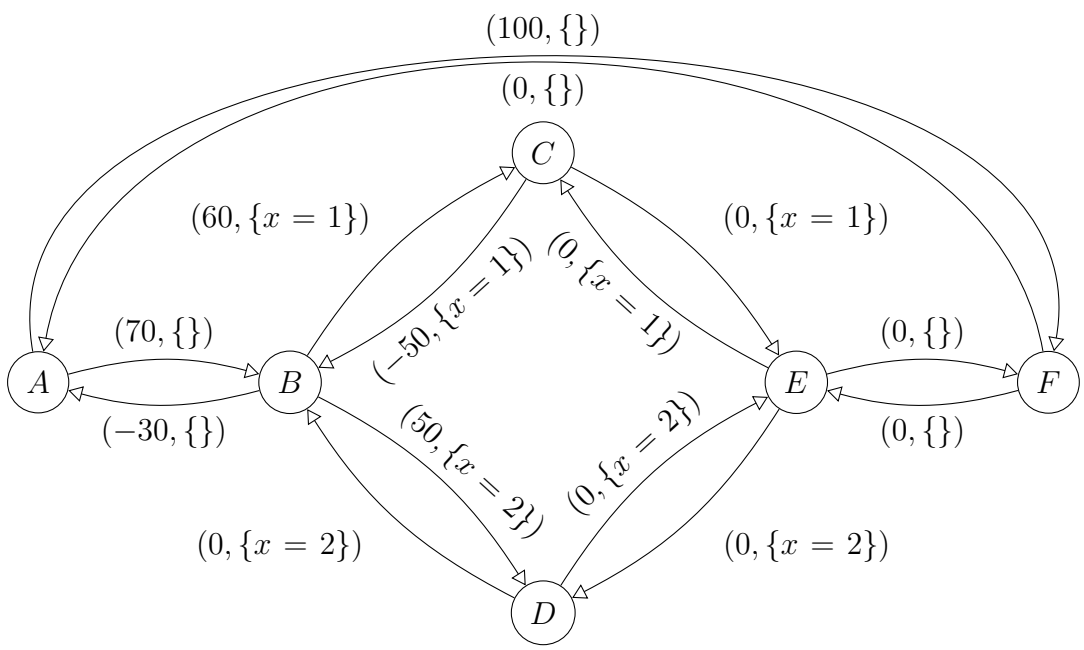
Drake's dispatcher relies on these two fundamental tests, developed by Muscettola, for whether an event is executable. Drake adds support for the compact data structures developed in Chapter 3, storing execution windows in labeled value sets and using the temporal constraints represented in a labeled distance graph. Drake also adds reasoning steps to allow it to consider the possible choices available and to select between them. This is similar to Tsamardinos's overall approach, but implemented with a conflict database. Finally, it adds support to search for and execute activities.

Algorithm 4.1 provides pseudo-code for the Drake's top level dispatcher, called DispatchLabeledDGraph. The dispatcher uses functions developed throughout this chapter. The input to the dispatcher is the compiled version of the input problem, and is specified by the list of events V, the labeled value sets representing the edge weights, W, and the conflict database, S. It also takes a specification of the activities, Act, which is defined fully in Section 4.5. Essentially, the activity structure specifies

which constraints of the original plan are activities, their environments, and some identifier of what physical activity it represents. The result of applying the algorithm is that either the events and activities are executed according to the specification of at least one of the component STPs or that an error is detected and signaled. We use the following example to walk through the dispatching process.

Example 4.1 Let us consider dispatching the labeled distance graph corresponding to the TPN in Figure 2-1, which is replicated in Figure 4-1. Although this graph has not been compiled into dispatchable form, which is generally a prerequisite for the dispatcher, it is simpler to draw and still allows us to walk through the algorithm. Recall that the edge (A, B) is the drive activity of this plan, which would be encoded in the *Act* data structure. □

Figure 4-1: The labeled distance graph corresponding to the TPN of Figure 2-1. All edges not drawn are implicitly valued $(\infty, \{\})$. The variable x denotes the choice and has domain $\{1, 2\}$.



The first step of the algorithm, which is on Line 2, initializes sets to hold the events that have been executed and events that are still waiting on activities. Also, the initial time is set to zero, arbitrarily, as is typical in the literature. Lines 4-7

Algorithm 4.1 The top level dispatching algorithm.

```
1: procedure DispatchLabeledDGraph(V, W, S, Act, Δt)
2:     V_exec, V_waiting ← ∅                                   ▷ Initialize event sets
3:     t ← 0
4:     for i ∈ V do                                            ▷ Initialize execution windows
5:         B_i^u ← (∞, {})
6:         B_i^l ← (−∞, {})
7:     end for

8:     (B, S, V_exec, v) ← ExecuteIfPossible(V, W, V_exec, S, B, V_start, 0)   ▷ Execute
   start event
9:     V_waiting ← V_waiting ∪ v

10:    while V ≠ V_exec do
11:        S ← CheckUpperBounds(V, W, V_exec, S, B, t)   ▷ Alg. 4.4, find violated
   upper bounds
12:        V_finished ← GetFinishedActivities()
13:        V_waiting ← V_waiting \ V_finished

14:        for i ∈ V \ V_exec \ V_waiting do                   ▷ Try to events
15:            (B, S, V_exec, v) ← ExecuteIfPossible(V, W, V_exec, S, B, i, t)
16:            V_waiting ← V_waiting ∪ v                       ▷ store starting activities
17:        end for

18:        t ← t + Δt                                          ▷ Increment time
19:        wait Δt
20:    end while
21: end procedure

22: procedure ExecuteIfPossible(V, W, V_exec, S, B, i, t)
23:
24:    if S_removed ← EventExecutable?(V, W, V_exec, S, B, i, t) then    ▷ Alg. 4.3
25:        B ← PropagateBounds(V, V_exec, W, S, B, i, t)                  ▷ Alg. 4.2
26:        (S, V_waiting) ← BeginActivities(V, V_exec, W, S, B, i, t)     ▷ Alg. 4.5
27:        V_exec ← V_exec ∪ i                                 ▷ Store execution
28:    end if
29:    return B, S, V_exec, V_waiting
30: end procedure
```

initialize the upper and lower bounds for all events to provide no restrictions on their execution times. For example, the start event, A, is given a lower bound of $(-\infty, \{\})$ and an upper bound of $(\infty, \{\})$, as are all the other events. The last initialization step is to execute the start event, A in our example, as shown on Line 8 by calling ExecuteIfPossible. Although the start event is executable by definition, this function is used to first determine if an event is executable and greedily execute it if so. The execution is greedy because this function immediately executes any event it proves is executable.

The function ExecuteIfPossible is responsible for selecting events to execute and actually schedule them. First, Line 24 calls EventExecutable?, developed in Section 4.3, which determines if the event is executable by applying the modified versions of Muscettola's criteria, described above. At any time, executing a particular event might be consistent with all the possible choices, none of the choices, or some of the choices, which function determines through operations on the conflict database. For example, if the start event A is executed at time $t = 0$, then B is not executable at time $t = 10$, because the lower bound induced by the edge weight $(-30, \{\})$ is not met. Therefore, this scheduling decision is not consistent with any of the choices because the bound that is not met has an empty environment, which subsumes all possible choices. If executing the event at the current time is not consistent with any of the choices, then it cannot be executed and the algorithm and moves on. In this example, B cannot be executed at $t = 10$, and the algorithm moves on.

If the scheduling the event at the given time is consistent with all remaining possible choices, then no conflicts are added and the event may be executed without making any commitments. For example, if the executive waits until $t = 40$, then B can be executed without removing any choices. However, if the scheduling decision is only consistent with some of the remaining choices, then conflicts are added to the database to represent those excluded choices, which the dispatcher removes from future consideration. For example, the executive might schedule event C out of turn, before B has executed, which it may only do if it creates a conflict for $\{x = 1\}$, signaling that the dispatcher has invalidated that choice.

If the event is deemed executable, ExecuteIfPossible continues with the steps required to actually execute the event. First, Line 25 updates the execution windows of all neighboring events by performing one-step constraint propagations using an algorithm developed in Section 4.2. When the start event A is executed in this example, the connected events, the end of the drive B and event F, would be updated.

Now the dispatcher can search for and execute activities that begin with the start event, A. Line 26 calls BeginActivities, developed in Section 4.5, which is responsible for finding any events that begin with the event currently being executed, and are consistent with the choices available. In this case, the dispatcher finds the drive activity, which must begin with the start event regardless of which choices the dispatcher makes. Drake tells the system to asynchronously begin the drive activity, using the smallest possible time duration of 30 time units, and to return event B when the drive is complete. Activities are executed asynchronously, so the dispatcher may continue to schedule events while this activity occurs, but adds B to the list $V_{waiting}$, because that event cannot be executed until the activity is done. Later, Line 12 performs a system call that polls whether any activities are complete, signaled by returning B. Line 13 removes the returned event B from $V_{waiting}$, allowing the system to execute it.

The final step of ExecuteIfPossible is to add the newly executed event to V_{exec} to indicate that is has been executed.

Returning to the top level function DispatchLabeledDGraph, the remainder of the function is a while loop that allows time to pass until all the events are executed. At each time step, several functions are run. First, Line 11 determines whether any choices have become invalid because of a missed upper bound on an event. One reason for this might be an unexpected delay in an activity that prevents an event from being executed on time. When an upper bound is missed, it may eliminate possible choices or may cause dispatch to fail. The function CheckUpperBounds is developed in Section 4.3. For example, if Drake failed to execute B within 70 minutes of executing event A, it has missed an upper bound that causes the execution to fail because the bound is necessary for every component STP. Then Drake checks for finished

activities as mentioned above. The block beginning on Line 14 searches through the events that might be executable, specifically, those that have not been executed and are not waiting on activities to finish, and executes them if possible. Finally, the time is incremented and the dispatcher waits for time to elapse before beginning again.

In this example, between times $t = 0$ and $t = 30$, no events may be executed, because only B has all its ordering constraints met, but it is still waiting for the drive to complete. If the activity completes at $t = 32$, then B is removed from $V_{waiting}$, and is scheduled at the same time step. Its execution time is then propagated to its neighbors. Then this sequence repeats until all the the other events are executed. Note that all events are executed, even though the two paths in this example are considered mutually exclusive in a TPN. This is acceptable because the activities, which specify real actions, are only initiated after checking the environments and choices for this exclusivity. Therefore, while the events for unselected paths become unconstrained and may be executed at arbitrary times, they remain within the dispatcher for book-keeping purposes only, and the dispatcher does not accidentally start any real actions because of them. For example, if event C is executed out of turn, this invalidates the option to collect samples. Therefore, when B is executed, the dispatcher does not initiate an that activity because its environment is inconsistent. See Section 4.5 for a more detailed development of the activity reasoning algorithms. Also note that a DTP does not encode exclusive execution paths so every execution must contain every event.

The remaining sections of this chapter fill in the details of this top level algorithm. The standard STP-based reasoning is described in two sections. First, Section 4.2 develops mechanisms for performing constraint propagation on labeled distance graphs, which is needed to compute the execution windows for events. Second, Section 4.3 uses those windows to determine when events may be executed. Third, Section 4.4 presents the tests for violated upper bounds. Finally, we develop the additional components required for activity dispatching.

4.2 Labeled Execution Windows

Drake requires a labeled analogue to the execution windows, developed by Muscettola, to dispatch events. In STP dispatcher, the execution windows are maintained as a single upper and lower bound on each event's execution time [10]. In a DTP, Tsamardinos computed and stored bounds independently for each component of STP [19]. Drake modifies these strategies to use labeled value sets, in order to compactly represent the bounds for all component STPs. These labeled execution windows are directly computable from the compact representation of the compiled STPs provided by the labeled distance graph. In an STP, a single value bound is computed by propagating an execution time through a weighted edge. In Drake, the compact form of the bound is directly computable by propagating an execution time through and edge of a labeled distance graph, which is a labeled value set itself, produces another labeled value set. This representation shows the dependence of the execution windows and the choices, which the dispatcher uses to determines if an event can be executed at a particular time.

As with the constraints expressed on edges, Drake only needs to maintain the tightest bounds for each possible choice, making them naturally expressible through our concept of *dominance*. The upper and lower bounds have different dominance conditions for their labeled value sets; the temporal reasoning Drake performs is only affected by the lowest known upper bound and the highest known lower bound, which is why the dominance functions are $<$ and $>$, respectively. The initial bounds are set as loose as possible, positive and negative infinity for the upper and lower bounds, respectively.

Definition 4.2 (Labeled Execution Windows) For a labeled distance graph G, Drake represents the times each event may be executed with labeled value sets B_i^l and B_i^u for the lower and upper bounds, respectively. For each event $i \in V$, B_i^l is a labeled value set with $f(a, a') \leftarrow (a > a')$ and B_i^u is a labeled value set with $f(a, a') \leftarrow (a < a')$. The bounds are collectively referred to as B. All bounds are initialized with $B_i^l = (-\infty, \{\})$ and $B_i^u = (\infty, \{\})$. □

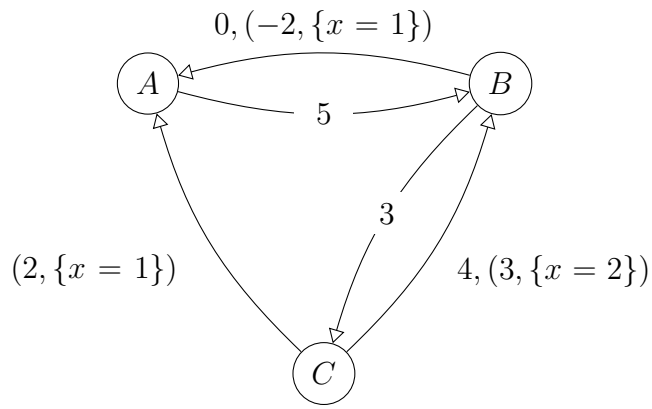
Example 4.3 An event might have $B^l = ((5, \{x = 1\}), (2, \{x = 2\}))$ and $B^u = ((10, \{x = 1\}), (4, \{x = 2\}))$. In this case, there are two possible execution windows. If the executive selects $\{x = 1\}$, the event may be executed in the window $[5, 10]$, otherwise if $\{x = 2\}$, then the window is $[2, 4]$. □

When an event is executed, Drake updates the execution windows of neighboring events, reflecting the constraints represented in the graph. In an STP, executed event times are propagated through outgoing edges to update the upper bounds of neighboring events and through incoming edges to update lower bounds [10]. Drake performs the same propagations, substituting labeled operations as necessary. Algorithm 4.2 performs this operation, updating the structure containing the execution windows, B, with the consequences of executing event i at time t. The other inputs specify the labeled distance graph, V, W, and S, and the current state of the execution, V_{exec} and B. To illustrate this algorithm in action, consider the scenario presented by the following example.

Example 4.4 Consider the compiled labeled distance graph from Example 3.26, computed in Chapter 3 and replicated in Figure 4-2. Assume that event A is the first event to execute, at $t = 3$, and PROPAGATEBOUNDS is called. We also assume that all the bounds begin with their initial infinite values. The upper and lower bounds for events A, B, and C are summarized in Table 4.1, before and after the function call, and are derived below. □

As first step of PROPAGATEBOUNDS, Line 2 sets the upper and lower bound for the executed event to be the execution time with an empty environment, $\{\}$, meaning that the execution time holds for all choices. In this example, since event A is executed at time $t = 3$, its upper and lower bound are both replaced with $(3, \{\})$. Next, Lines 3-6 loop through every other non-executed event, updating the lower and upper bounds. The addition or subtraction operations are carried out with LABELEDBINARYOP, as appropriate for labeled value sets. MERGECANDIDATES from Algorithm 3.5 ensures that all of the bounds represented are useful for some possible execution, according

Figure 4-2: The filtered, labeled distance graph from Example 3.26.



to the conflict database S. Note that the domination functions Drake uses to merge values differs for upper and lower bounds, as specified in Definition 4.2.

In our example, the for-loop block updates the upper and lower bounds of events B and C in turn. First consider updating the lower bound of event B. The lower bound is updated with the edge weight on (B, A), subtracted from the execution time. We can perform the computation of the new bounds as

$$(3, \{\}) - ((0\{\}), (-2, \{x = 1\})) = ((3, \{\}), (5, \{x = 1\})) \tag{4.4}$$

This new labeled value set is merged with the existing one, $(-\infty, \{\})$, by the function MergeCandidates, replacing the old value with the new one. The newly computed pair $(3, \{\})$ replaces the old negative infinity because it has the same environment and is strictly a tighter constraint. The value of five dominates three, but the environment is more specific, so the five does not allow us to prune the three. The upper bound of B is computed as the sum of two values with empty environments, $\{\}$, producing the value of $(8, \{\})$ that replaces the old infinity.

Only the lower bound of event C is updated during this function call, because there is no edge (A, C) to update the upper bound. The lower bound adds a new pair $(1, \{x = 1\})$ to the labeled value set, but does not remove the old value $(-\infty, \{\})$, because the environment $\{x = 1\}$ does not subsume the empty environment.

Theorem 4.5 (Compact Execution Windows) *The labeled value sets stored in*

Algorithm 4.2 Propagate bounds for an executed event.

```
1: procedure PropagateBounds(V, V_exec, W, S, B, i, t)
2:     B_i^l = B_i^u = (t, {})
3:     for j ≠ i, j ∈ V \ V_exec do
4:         B_j^l ← MergeCandidates(B_j^l, B_i^l − W_ji, S, >)    ▷ Alg. 3.3 and Alg. 3.4
5:         B_j^u ← MergeCandidates(B_j^u, B_i^u + W_ij, S, <)    ▷ Alg. 3.3 and Alg. 3.4
6:     end for
7:     return B
8: end procedure
```

Table 4.1: The execution windows before and after the update for Example 4.4.

Bound	Before	After
B_A^l	$(-\infty, \{\})$	$(3, \{\})$
B_A^u	$(\infty, \{\})$	$(3, \{\})$
B_B^l	$(-\infty, \{\})$	$((5, \{x = 1\}), (3, \{\}))$
B_B^u	$(\infty, \{\})$	$(8, \{\})$
B_C^l	$(-\infty, \{\})$	$((-\infty, \{\}), (1, \{x = 1\}))$
B_C^u	$(\infty, \{\})$	$(\infty, \{\})$

B and computed with Algorithm 4.2 provide a compact representation for the execution windows stored on the component STPs. This representation provides the same information as if the STPs were dispatched individually. □

We can prove this by arguing that the labeled distance graph is a compact representation of the constraints, and that our operations on labeled value sets are correct. Therefore, the labeled propagations derive the correct bounds.

Proof Theorem 3.21 shows that the labeled distance graph is a compact representation of all the constraints of the compiled, component STPs created by Tsamardinos's approach. Therefore, our propagation step begins with all the necessary constraints. The time of execution is correctly given an empty environment because the execution time is fixed without requiring any assumptions about which choices are selected. The propagation computation performs addition or subtraction on the labeled value sets, which is proved correct by Theorem 3.20, calculating the same candidates as performing the simple addition or subtraction for each component STP. Finally, the candidate bounds are stored in labeled value sets, which are lossless by 3.18. The

bounds of the component STPs are recoverable by querying the labeled value sets with the complete environments associated with the component STPs, so the bounds structures are a complete, compact representation of the execution windows. ■

This algorithm provides a way to compute and store execution windows using Drake's compact representation of the distance graph. Labeled value sets allow us to avoid representing the bound independently for each component STP, but instead we can compactly record the dependence of the bounds on choices. Drake's propagation algorithm is essentially the same as Muscettola's STP one-step propagation algorithm, except that we have replaced the scalar values with labeled value sets and substituted in the proper operations. Furthermore, we have proved that they store the same information as if the STPs were executed independently, as in Tsamardinos's work.

4.3 Selecting Events to Execute

This section develops the algorithm that Drake repeatedly calls to determine if a particular event is executable at the current time, given the compiled form of the input plan and the execution sequence thus far. The execution criteria that Drake uses are essentially Muscettola's criteria for executing events in STPs, combined with Tsamardinos's technique for selecting choices. These methods from the literature are adapted to our compact, labeled representations and presented in the function EventExecutable? in Algorithm 4.3.

Within the STP dispatching framework, Muscettola showed that an event is executable if the current time is within its execution window and all its predecessors have also been executed. These two criteria remain intact in Drake. However, Drake can select between different possible component STPs at run-time, which might be contradictory, such that no execution can satisfy all of them. Typically, as an execution unfolds, Drake must make incremental commitments, narrowing from a large array of initially feasible component STPs, down to one, or a few, that it actually executes and satisfies all the constraints of. It is possible that Drake might reach the end of an execution having satisfied the requirements of several component STPs, having some

choices that are unresolved, but we consider that a happy coincidence the system does not care about. We defined correctness as satisfying at least one component STP and there is no apparent benefit from satisfying more than one component STP simultaneously, although it is not problematic either. To make these decisions while guaranteeing correctness, we use Tsamardinos's strategy. Drake is allowed to execute an event at any time that time is consistent with at least one of the remaining component STPs. After scheduling the event, the component STPs that are inconsistent with this scheduling are removed from consideration by creating conflicts. If only one component STP remains, Drake must follow it exactly. If, because of some external event or unexpected delay all remaining STPs are invalidated, then the execution has failed and Drake throws an error, requiring re-planning at a higher level.

The prior literature provides sufficient guidance for how to make the execution decisions at run-time; we simply need a strategy for performing this reasoning correctly and efficiently with our compact encodings. Identifying which remaining component STPs satisfy Muscettola's event execution requirements is complicated by the compact encoding, because each component STP's execution windows are not explicitly stored. Therefore, we develop an algorithm that tests the execution windows directly on the compact encoding, using the conflict database to determine whether the scheduling decision is consistent with any remaining STPs.

Example 4.6 Let us return to Example 4.3, where we consider a single event. If the executive selects $\{x = 1\}$, the event may be executed in the window $[5, 10]$; otherwise if $\{x = 2\}$, then the window is $[2, 4]$. These two execution windows are mutually exclusive, so dispatching this event requires the executive to make a decision between them. For example, executing the event at time $t = 4$ satisfies the $\{x = 2\}$ window, but violates the lower bound for the window corresponding to $\{x = 1\}$, thus invalidating any possibility for selecting that choice. The executive can record this fact by creating a conflict for $\{x = 1\}$. □

This example illustrates that the dispatcher narrows the possible choices at run-time, by creating conflicts when it violates constraints. There are two types of con-

straints that might be violated: activation constraints that specify a strict ordering of event executions and execution windows. For example, the rover has an ordering constraint indicating that the end of the drive, B, may not execute before the start of the drive, A. From the example above, executing the event outside of the window $[2, 4]$ violates a constraint. If Drake violates a constraint at run-time, the environments of those violated constraints become conflicts. For instance, violating the window $[2, 4]$ from the example above requires creating the conflict $\{x = 2\}$, because this is the environment for the violated constraint. The conflict exactly summarizes which component STPs are invalidated by that execution. Instead of finding the STPs where a particular execution is allowed, Drake determines whether it can create all necessary conflicts for that execution without invalidating all possible environments. Determining if an event is executable, then, requires collecting the environments of activation constraints and execution windows that would be violated by executing the event, and testing whether they can all be removed as potential options.

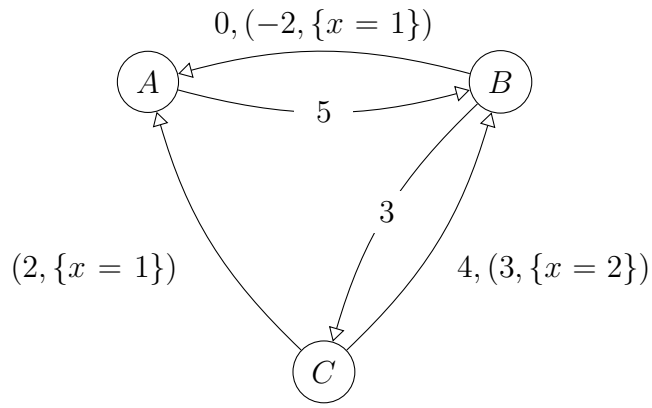
Example 4.7 We continue the execution process for Example 4.4, where the labeled distance graph is replicated in Figure 4-3. The execution windows, after event A is executed, are summarized in Table 4.1. Assume that A was executed at $t = 3$. If we executed B at $t = 4$, that would violate the lower bound $(5, \{x = 1\})$ on B. Recall that this constraint states that if $\{x = 1\}$, then the execution time of B must come later than $t = 5$. Executing B at time $t = 4$, therefore, implies that the dispatcher must not select $\{x = 1\}$, which we note by creating a conflict.

We may schedule this event and create the corresponding conflict, if doing so leaves us at least one possible option. If there is not at least one remaining option, then the scheduling decision requires making all remaining component STPs inconsistent, causing the execution to fail. In this case, $\{x = 2\}$ remains a viable option. Therefore, we are free to execute B at $t = 4$ and create the conflict. Since $\{x = 2\}$ is the only remaining option, the dispatcher needs to satisfy all associated constraints.

Instead of executing B at $t = 4$, we might consider a later time, $t = 9$. However, every possible choice requires the upper bound of 8, so waiting until a time later than $t = 8$ would invalidate every possible choice. Therefore, the dispatcher cannot select

that time. □

Figure 4-3: The filtered, labeled distance graph from Example 3.26.



Algorithm 4.3 performs the complete task of testing whether an event is executable at the current time. The function EventExecutable? is called on an event just before it might be executed, and asks whether the dispatcher may execute the event at the current time. Its inputs are the weighted graph, the list of executed nodes, the constraint database, the bounds, the event in question, and the current time. The output is either true, signaling that the event should be executed at the current time and that the conflict database has been updated accordingly, or false, signaling that the event may not be executed yet. During the discussion of the pseudo-code, we reconsider Example 4.7, determining whether event B may be executed at time $t = 4$.

The first phase of the algorithm is to identify all constraints that would be violated by executing event i at time t. Line 2 initializes $e_{violated}$, a set that will hold the environments of violated constraints. Then, Lines 3-8 search through the upper bounds on the event i and store any environments for bounds lower than t. For example, the lower bound on B is $(8, \{\})$, which is not violated by executing B at $t = 4$. Similarly, the following block, Lines 8-12 perform the same operation on the lower bounds. In our example, this process finds that while the bound $(3, \{\})$ is not violated, $(5, \{x = 1\})$ is, because executing B at time 4 is sooner than the lower bound. Therefore, the environment $\{x = 1\}$ is added to the set of violated environments.

To complete the first phase, Lines 13-19 search for negative outgoing weights from i to non-executed events, as these imply strict ordering constraints that are not currently met. For example, B requires that A is executed first if $\{x = 1\}$, because of the negative value $(-2, \{x = 1\})$ on the edge (B, A). However, A was executed, hence no constraint is violated.

Having collected all the labels of violated constraints, the second phase of the algorithm removes from future consideration all component STPs that contain these constraints. Line 20 queries the conflict database to determine if the environments can become conflicts without invalidating all possible environments. If that returns true, then the conflicts are created and the algorithm returns true, indicating that the event should be executed. Otherwise, no action is taken and the algorithm returns false. Completing our example, $e_{violated}$ has only the environment $\{x = 1\}$. No conflicts have been created yet, and there is another possible option if $\{x = 1\}$ is a conflict, so ConflictsPossible? returns true on Line 20. Therefore, the dispatcher creates a conflict for environment $\{x = 1\}$ on Line 21, and then commits to scheduling the event B at $t = 4$ by returning true. This action commits the dispatcher to select the only other option, $\{x = 2\}$.

To illustrate that empty environments correspond to universally mandatory constraints, assume a situation exists, in which the set $e_{violated}$ includes the empty environment, $\{\}$. By definition, the empty environment subsumes all possible environments, so making it a conflict necessarily invalidates all possible environments. Therefore, the dispatcher is never allowed to violate a constraint with an empty environment.

This method is the core reasoning method used to schedule events, as the dispatcher can repeatedly query whether the events are executable as time passes and execute each one when the function indicates they can. We need to prove that following this algorithm produces correct executions.

Theorem 4.8 (Event Selection) *The algorithm for* EventExecutable? *only indicates that the input event is executable if it is executable in one of the consistent component STPs.* □

Algorithm 4.3 Determine if an event is executable.

```
1: procedure EventExecutable?(V, W, V_exec, S, B, i, t)
2:     e_violated ← {}
3:     for (a_j, e_j) ∈ B_i^u do                          ▷ Test upper bounds
4:         if a_j < t then
5:             e_violated ← e_violated ∪ e_j
6:         end if
7:     end for

8:     for (a_j, e_j) ∈ B_i^e do                          ▷ Test lower bounds
9:         if a_j > t then
10:            e_violated ← e_violated ∪ e_j
11:        end if
12:    end for

13:    for j ∈ V \ V_exec do                              ▷ Test activation
14:        for (a_k, e_k) ∈ W_ij do
15:            if a_k < 0 then
16:                e_violated ← e_violated ∪ e_k
17:            end if
18:        end for
19:    end for

20:    if ConflictsPossible?(e_violated) then             ▷ Test for remaining solutions
21:        AddConflicts(e_violated)
22:        return true
23:    else
24:        return false
25:    end if
26: end procedure
```

PROOF The labeled distance graph and the execution windows are a condensed version of all the weights and execution bounds of all the component STPs, as proved by Theorem 3.21 and Theorem 4.5. The algorithm finds all the violated constraints and collects their environments. If the event is executed at time t, then the environments of the violated constraints are conflicts, because every component STP whose complete environments is subsumed by one of the violated environments contains a violated constraint. The function EVENTEXECUTABLE? only returns that the event is executable if constraint database reports that there are still valid complete environments, meaning that there is at least one STP where the execution decision is legal. ■

Note that this theorem specifies that Drake replicates the dispatching decisions of an STP dispatcher, indicating that Drake inherits the guarantees that an STP dispatcher can successfully execute a dispatchable STP.

There is an important design decision implicit in this algorithm, however, in that it generally commits to executing an event at the earliest time that it is possible to do so. Although Tsamardinos provides methods to determine whether a delay is possible or when the event might be executed in the future, Drake simply executes events as soon as possible to simplify the activity algorithm, as discussed in Section 4.5. While we do not explore the possibility here, it should be possible mirror Tsamardinos's reasoning about future execution times.

This section presented the algorithm for determining if an event may be executed at the current time. It closely follows the strategies provided by Muscettola and Tsamardinos for performing the reasoning, while adapting the steps to our representations. In short, an event is executable if the environment for every constraint and bound violated by the proposed execution can become conflicts, without invalidating all possible environments.

4.4 Finding Violated Bounds

By selecting events to execute, the dispatcher is allowed to directly violate the constraints within some component STPs as long as there are alternative STPs where no constraints are violated. However, in a dynamic execution system, there may be unexpected delays that violate constraints. The dispatcher needs to identify these delays and appropriately handle the violated constraints. As during event selection, violated constraints create conflicts to represent the invalid STPs. However, since the dispatcher is not necessarily in control of these violated constraints, it is possible that all remaining component STPs would be invalidated, thus signaling a failure.

Specifically, as time passes, the upper bound of some event might pass, signaling an invalidated STP or possibly that the execution has failed. At every time step, Drake needs to check for upper bounds that have been violated and prune them from future consideration. The function CheckUpperBounds, called every iteration from the top level dispatcher and presented in Algorithm 4.4, performs this test. Note that the code is similar in structure to the algorithm for EventExecutable?.

Algorithm 4.4 Find and prune violated upper bounds.

```
1:  procedure CheckUpperBounds(V, W, V_exec, S, B, t)
2:      e_violated ← {}
3:      for i ∈ V \ V_exec do
4:          for (a_j, e_j) ∈ B_i^u do                    ▷ Test upper bounds
5:              if a_j < t then
6:                  e_violated ← e_violated ∪ e_j
7:              end if
8:          end for
9:      end for

10:     if AddConflicts(e_violated) then                 ▷ Test for remaining solutions
11:         signal failure
12:     else
13:         return
14:     end if
15: end procedure
```

Lines 3-9 searches through the list of non-executed events and look for upper bounds that have been violated by the passing of time. The environments for any

violated bounds are collected to create conflicts, in a nearly identical fashion as in EventExecutable?. The crucial difference from EventExecutable? occurs at the end of the function; these constraints have already been violated by the passage of time and the inaction of the dispatcher, hence there is no decision about whether or not to proceed in this fashion. Therefore, AddConflicts is called immediately on $e_{violated}$, without testing whether those conflicts make all the complete environments inconsistent. If the new conflicts invalidate all complete environments, the algorithm must signal that the dispatch has failed, shown on Line 11. Otherwise, it returns and dispatch continues.

Example 4.9 Consider Example 4.7 again, where event A was executed at $t = 3$. Event B has an upper bound $(8, \{\})$. If the dispatcher waited until $t = 9$ without executing event B, it would discover the failure when it called CheckUpperBounds at that time-step. The label of the violated bound is an empty environment, which subsumes every possible complete environment, invalidating all of them. Therefore, there are no remaining consistent STPs, and there is no possible execution. □

This section presented a simple, but important addition to the dispatching procedure. Although this additional check for missed execution windows is not necessary in a fully controllable world, it provides a crucial feedback mechanism for real world applications, in which there may be unexpected delays in the system. This check allows Drake to notice that a choice is no longer valid and to switch to alternate choices if some alternate is possible, or to notice plan failure as quickly as possible, allowing a re-planning step to determine a new plan.

4.5 Dispatching Activities

Dispatching a Temporal Plan Network with activities requires different temporal semantics than Simple Temporal Problems, because an activity's duration must be set when it begins. Within a STP, activities are typically modeled with a start and an end event. A STP assumes that all time points can be instantaneously scheduled if

its requirements are met, giving the dispatcher complete flexibility to execute the end event of a process once the minimum time bound and other requirements are met. This assumption is not realistic in all cases; often the duration of physical activities need to be determined when they start and cannot be terminated arbitrarily at run-time. For example, in the rover example, the drive activity cannot arbitrarily end with the rover at the goal location, but must be scheduled in advance. In light of the different semantics, this section proposes a method for modifying a STP based dispatcher to handle this case. Specifically, we develop a function BeginActivities, which is called when an event is executed, and whose purpose is to select the durations of activities at their start time. Essentially, we select the durations following a strategy of "hurry up and wait," which selects the shortest possible duration and then inserts waits as necessary.

The following example highlights the issue with STP activity semantics.

Example 4.10 The running rover example begins with a drive and then might include a period in which the rover charges the battery. It is quite reasonable that once the rover begins charging its batteries, it could stop charging on demand, thus meeting the STP semantics. The drive activity, however, is less flexible. If there is a pre-determined destination, the flexibility in the drive activity implies that the rover might be able to go faster or slower. However, it is not reasonable to assume that the drive activity can be terminated at any time. For example, say that the rover set out at a speed that allows it to arrive in about 60 minutes. After 30 minutes the temporal constraints would be met, so under the STP model the dispatcher could require the event to end immediately, even though the rover is only halfway to its destination. Although the STP dispatcher is allowed to instantly execute the end event, for this to correspond to the physical world, the rover would have to teleport to the end location. To be physically meaningful, the rover needs to plan ahead what the drive duration should be and set the drive speed accordingly. □

In early research on STP dispatching, Vidal refers to the two types of controllable intervals as *End Controllable* and *Begin Controllable*, denoting whether the

dispatcher can select the duration at the end of the interval or whether it must do so at the beginning, respectively [21]. Later on, STP dispatchers adopted end controllable durations throughout, but many physical systems can only be reasonably modeled with begin controllable durations. Temporal Plan Networks use activities to model physical processes, that typically require the begin controllable semantics [8]. Drake only allows begin controllable activities, but it would be simple to schedule end controllable activities without the advance duration selection.

To formally specify the activities in a plan, we define a data structure.

Definition 4.11 (Activities) The activities of a plan are specified in *Act*, a list of activity specifications. Each activity is specified by one item, *act*, which has fields l and u for the lower and upper bound, e for the environment, and *primitive* for the actual activity to execute. There are also fields for the *start* and *end* events. These activities are separately handled as simple interval constraints and are represented in the labeled distance graph along with all the other constraints of the STP. □

Example 4.12 In the rover example, the *Act* data structure would specify that the first activity specifies a drive, with duration $[30, 70]$, and an empty environment, since it must always happen. □

With this definition in place, we propose a method for adapting STP based dispatchers that allows the plan to follow a "hurry up and wait" strategy. When the duration must be selected in advance, providing tight synchronization on the ends of activities is difficult, without scheduling events in advance, giving up flexibility. In an STP, nothing stops us from scheduling events ahead of time, thus allowing us to pick the duration of the activity at its beginning, since all the events are fully controllable. However, we want Drake to perform dynamic dispatching so that the executive could be robust to minor disturbances. Selecting the duration ahead of time removes that flexibility for activities. Unfortunately, the STP formalism cannot simultaneously provide flexible execution and advance scheduling, so we slightly relax the definition of a correct execution.

Therefore, we separate the end of the actual physical process of the activity and the execution time of the end event. This allows us to select the duration of the actual activity when the start event is executed, without giving up flexibility on the execution time of the end event. Once the activity is complete, the end event of the activity is executed as soon as is feasible. To make this possible, we allow the dispatcher to insert a delay between the end of the actual activity and the end event, thus, relaxing the constraint in a way that seems reasonable for many cases. This strategy is always self-consistent, because aiming for the shortest possible execution means that any updates to the execution windows that happen during the activity can only require that the end be delayed. This delay is exactly what we have empowered the dispatcher to do. Now we give the formal modification to the execution requirement.

Definition 4.13 (Activity Execution) In a TPN, consider an activity between events X and Y, specifying that the duration $Y - X \in [l, u]$, where $u \geq l \geq 0$. The activity is correctly executed if the $t_{exec} \in [l, u]$ and $t_{exec} + t_{wait} \in [l, u]$, where t_{exec} is the time from the beginning of the start event to when the actual activity ends and t_{wait} is the duration between the end of the activity and when the end event Y executes. □

The lower bound of the activity is required to be positive so that it provides a strict ordering, stating that X is the deterministic start event of the activity and Y is the end event. The definition states that the activity duration must be within the original bounds and also that the duration plus any wait inserted must lie within the original bounds.

Example 4.14 Consider the drive activity from the rover example, which has temporal bounds $[30, 70]$. If the drive actually takes 40 minutes, the dispatcher may wait up to 30 minutes to execute the event that signals the end of the drive. However, the drive cannot take 20 minutes and then be padded with a 10 minute wait because then the activity itself would not have been within the original constraint. □

With this relaxed definition, we can prove a simple, method for selecting the execution time of activities.

Theorem 4.15 (Dynamically Selecting Earliest Activity Completion) *Consider a dispatcher committed to executing an activity between events X and Y, specifying that the duration $Y - X \in [l, u]$, where $u \geq l \geq 0$. Then, without loss of generality, the dispatcher may always select $t_{exec} = \max(l, \textit{lower bound}(Y) - t_{curr})$ when X is executed. If the later terms are not available, omit them from the maximization.* □

PROOF If the plan is dispatchable and the activity is necessary for the completion of the plan, when $X = t_{curr}$, then by definition of the compiled form and the dispatching execution windows $Y - t_{curr} \geq \max(l, \text{lower bound}(Y) - t_{curr})$. Although parallel threads may change the bounds on Y, the lower bound can only be raised during execution, meaning that this strategy always produces a duration that is too short, allowing the dispatcher to insert waits correctly. Therefore, the theorem's choice of t_{exec} does not remove any flexibility from the plan and Y may be scheduled according to the STP based dispatcher. The maximization step is necessary because the actual edge weight representing the activity might have been pruned from the labeled distance graph. This step ensures that it is considered correctly. ■

This theorem specifies that the dispatcher always chooses the shortest activity duration. We can only suggest that while not applicable to all situations, proceeding as quickly as possible is a reasonable choice. The proof requires that the activity must be mandatory because, otherwise, the dispatcher might be allowed to eliminate the choice requiring the activity, before the activity ends. If the activity is part of a choice, the dispatcher can satisfy this requirement by committing to the choice when the activity begins. Normally the dispatcher only commits to choices by discarding intervals, but when the executive selects a choice that begins an activity, it has naturally committed itself.

Example 4.16 Consider a pair of events X and Y with an activity between them with duration constraint $[5, 10]$. If X is executed at time $t_{curr} = 4$ and at that time Y has a lower bound of 11, then the duration between them must be at least seven. Therefore, we can minimize the wait time by starting the activity with length seven. □

This technique for selecting execution times can be summarized in Algorithm 4.5. Its inputs are the compiled problem and the execution windows. It outputs the revised conflict database and any events that are the end of activities that have started. This function may also begin some activities. This function is called when event i is executed, at time t. The function loops over all activities that begin with this event, on Line 3.

Essentially, the algorithm determines if each activity should execute, and if so, how long it should take. Line 4 tests that the environment is still valid. If so, the activity can be executed and the first step is to commit to its environment on Line 5 by calling CommitToEnv. For example, when beginning an execution of the rover problem, the dispatcher would see that the empty environment attached to the drive activity is still valid, and that committing to it has no effect. In contrast, beginning the charge activity when B executes requires committing to $\{x = 2\}$.

Next, the function determines the correct activity execution time. The execution time is either the lower bound of the activity from the original problem or the least restrictive valid lower bound on the end event, whichever is greater. This is computed on Lines 6 - 12. In the rover problem, in general, the execution duration will be the lower bound of the activity, 30 time units. No other edges could tighten the lower bound of the end event of the drive, so we need only consider the duration of the activity.

Line 13 calls the function BeginActivity, which tells the system to actually execute activity *act* with a duration of t_{exec}, and to return event *act.end* to the top level dispatcher when the activity completes, to release the end event for execution. Finally, Line 14 marks that the event *act.end* is the end of an ongoing activity and that the dispatcher must wait for it to complete. Forcing the dispatcher to actually wait for the completion of the activity, before executing the end event, is necessary for the dispatcher to be reactive to delays in the real world, although without a model of uncertainty, there are no guarantees that unexpected delays will not cause the plan to fail. Here we assume that this activity is the only one ending at this event.

To recapitulate, this strategy for handling activities, intuitively called "hurry up

Algorithm 4.5 A function to begin activities starting with a given event.

```
1:  procedure BeginActivities(V, V_exec, W, S, B, i, t, Act)
2:      V_waiting ← ∅

3:      for acts.t.(act.start = i) ∈ Act do
4:          if EnvironmentValid?(S, act.e) then
5:              CommitToEnv(S, act.e)                          ▷ Commit to activity

6:              t_exec ← ∞

7:              for (a, e) ∈ B^l_act.end do                    ▷ Find the loosest lower bound
8:                  if EnvironmentValid?(e) then
9:                      t_exec ← min(t_exec, a)
10:                 end if
11:             end for

12:             t_exec ← max(t_exec − t, act.e)

13:             BeginActivity(act, t_exec, act.end)            ▷ Start the activity
14:             V_waiting ← V_waiting ∪ act.end
15:         end if
16:     end for
17:     return S, V_waiting
18: end procedure
```

and wait," is one simple method for modifying the STP framework to handle activities with durations that must be set at the activity's start time. Although it does remove some guarantees about the synchronization of the execution, it is self-consistent and works without requiring a vast departure from the STP literature. The event selection algorithm and the activity scheduling algorithm are the remaining elements required for the dispatching algorithm. The next chapter completes our discussion.

4.6 Conclusion

This chapter completes our presentation of Drake's deterministic dispatching algorithm. When paired with the compilation techniques from Chapter 3, we have provided sufficient tools to dynamically dispatch a TPN, or DTP, as desired by this work. The dispatching algorithm handles the reasoning on temporal elements and the choices available by efficiently storing the constraints in minimal dominant labeled value sets. This chapter provided algorithms for temporal constraint propagation, event selection, constraint updates, and activity selection. The dispatcher handles activities and simplifies reasoning by greedily selecting the fastest options available. The greedy selection mechanism does restrict the solutions that the dispatcher can create.

While the deterministic STP compilation and dispatch algorithms introduce the important concepts and innovations behind Drake, we still need to introduce temporal uncertainty into the model. The next chapter follows the same themes, taking advantage of the compact labeled representation, while making relatively simple updates to the prior work, giving Drake the capability to handle explicit models of uncertainty.

Chapter 5

Plans with Choice and Uncertainty

Morris et al. demonstrated that the great strength of compilation and dispatchable execution is its ability to provide explicit guarantees about whether it can successfully execute a plan, even if some of the durations are not controllable by the executive [9]. Furthermore, reasoning about set-bounded uncertainty in this model is possible in polynomial time. Specifically, this prior work demonstrated that Simple Temporal Problems can be extended to include bounded uncertain durations in the problem specification, creating Simple Temporal Plans with Uncertainty (STPU). Then the executive can analyze the problem in order to determine whether it can execute the plan correctly, thus guaranteeing robustness to the modeled uncertainty in the real-valued execution times of events. This chapter outlines techniques to replace STPs with STPUs as Drake's underlying temporal model, allowing Drake to dynamically select between a family of STPUs, thus providing a guarantee of robustness to the uncertain outcomes for the component STPUs.

Adding explicitly modeled uncertainty into the problem is another way of handling activities and other uncertainties that arise when the executive interacts with the real world. While the intention of this thesis is to develop an executive that is robust to disturbances, the approach described in Chapters 3 and 4 provide robustness that is unpredictable. While this strategy is less fragile than a static schedule, it is difficult to know what range of uncontrollable delays it can handle for any particular problem. On the other hand, an explicit model of uncertainty directly allows the executive to

prove robustness at compile-time. Providing this guarantee requires the executive to be extremely conservative because it assumes that every uncontrollable duration resolves in the least favorable way. This conservatism is evident in both the limited scope of problems that are found feasible and in the execution time selected. However, as a designer of autonomous systems, it is a valuable tool to be able to specify particular uncertain outcomes and have a guarantee that the dispatcher cannot fail because of those outcomes.

Example 5.1 (Rover Example with Uncertainty) To illustrate the utility of uncertainty in dynamic execution, we can make the drive activity of the rover scenario of Example 1.1 uncontrollable. This modeling choice makes sense because at the outset of the drive, the rover does not know how many obstacles it will encounter or how quickly surface conditions will allow it to drive. Therefore, we indicate that any outcome in the range $[30, 70]$ is possible and must be handled by the system. The charging option provides enough flexibility to meet the deadline constraint regardless of the outcome of the drive duration. This is because it allows any duration from 0 to 50 minutes and can fill any duration remaining before the deadline of 100 minutes. In contrast, sampling is only acceptable if the drive is short, because sampling takes at least 50 minutes and the drive might take 70, which does not fit into the 100 minute deadline. □

This example illustrates the approximation Drake makes: instead of compiling the DTPU as a whole, Drake flexibly chooses between options inducing consistent component STPUs. In this case, Drake would discard the option to collect samples at compile time, conservatively restricting its options. This solution is somewhat limited, because collecting samples is not totally useless and need not be discarded completely; if the drive does resolve quickly, it is feasible, and charging provides an acceptable backup if the drive is slow. Another executive might be able to take advantage of the complementary nature of these two options, reducing the conservatism at compile-time, but we do not explore this idea further. Instead, our aim is to leverage the compact representations developed for Drake to simply handle families of STPUs.

Labeled value sets provide a compact and efficient representation for reasoning about related STPs. This chapter extends Drake to reason about families of related STPUs, facilitating an approximate dynamic controllability and dispatching algorithm for plans with uncertainty. As in the deterministic case, our strategy is to use labeled value sets to efficiently represent the interaction of the choices of the input plan with the temporal constraints. The process is remarkably simple: the constraint reasoning steps required for uncontrollable durations, as provided by prior work, are augmented with environment processing steps and the dispatcher is enhanced to handle the new type of constraints created by the STPU compilation process, that is, *conditional constraints*. These are constraints that specify an edge weight until an uncontrollable event executes, but constraint is removed once the uncontrollable event executes. Conditional constraints are required for the correct execution of STPUs. The result is a compilation algorithm for uncontrollable problems with choice and a dispatcher that can dynamically select the choices while respecting the uncontrollable durations.

This chapter begins by introducing some of the challenges of reasoning about uncontrollable durations in Section 5.1. Next, we define the uncertain versions of the possible input problems and our representation for reasoning about them in Section 5.2. Then, Section 5.3 outlines the compilation technique. Next, Section 5.4 describes the minor additions to dispatching required for the uncontrollable durations. Finally, Section 5.5 provides some concluding thoughts.

5.1 Background on Simple Temporal Problems with Uncertainty

Before giving our detailed approach, we provide an intuitive overview of Drake's compile-time and run-time processes. An STPU compiler is similar to a STP compiler, except that the compiler must prove that at run-time, the dispatcher never needs to restrict the execution time of the uncontrollable durations. Instead, the

dispatcher must be able to solve the STPU regardless of what value nature selects for the uncontrollable durations. At run-time, the dispatching algorithm is nearly identical to the STP dispatcher, except that handling the uncontrollable durations requires an additional type of constraint, called a *conditional constraint*, and which requires a minor addition to the dispatching routine.

Informally, a STPU is an STP where some of the constraints are marked as representing uncontrollable durations. This means that after the start of the constraint is executed, the end event executes automatically sometime during the feasible duration, but is outside the control of the executive. We illustrate the types of reasoning required at compile-time with the following example.

Example 5.2 Again, consider converting the drive activity of the rover example into an uncontrollable duration. There is a simple temporal constraint between the start and end events of the activity, with value $[30, 70]$, except that this constraint is marked as an uncontrollable duration. Once the start event executes, the activity begins, and may end at any time within the bounds of the activity's duration. This outside of the executive's control. In this case, the drive may take anywhere from 30 to 70 minutes. The executive only observes at each step whether the activity has finished and is given no estimates of when that will occur. To execute this uncontrollable duration correctly, the executive must not restrict the times when the end event may execute beyond the restriction imposed by the $[30, 70]$ constraint. This is because the executive is prohibited from influencing the outcome of the duration and therefore the executive is not able to enforce such a tightening.

There are two possible ways the executive might restrict the execution time of the end event. First, at compile time, computing the dispatchable form of the graph might tighten the weights of the edges from $[30, 70]$ to some tighter value. Arbitrarily, say the edges representing this constraint are tightened to $[35, 60]$; this modification is not allowed because the executive cannot dictate this duration, and cannot guarantee that the duration will fall within these bounds at run-time. However, if this tightening is an unavoidable consequence of the constraints of the plan, then the plan is infeasible according to the requirements of dispatchable execution for this model of uncertainty.

Checking for this type of problem is called testing for *pseudo-controllability* [9].

The second type of restriction an executive might impose is tightening the execution window of the end event at run-time. Assume that the start of the drive occurs at $t = 10$. Propagating this execution time through the activity's constraint leads to the conclusion that the end event must occur in the interval $[40, 80]$. If some other propagation attempted to tighten this window, for example closing the window to $[40, 70]$, that would signal another unacceptable restriction, although this type happens at run-time. To determine that a problem is *dynamically dispatchable*, meaning that the executive can successfully execute the plan with uncertainty, the compilation process must prove that neither of these types of restrictions on the execution of uncontrollable durations can occur. □

Section 5.3 defines and provides algorithms for these reasoning steps.

The dispatch algorithms are largely similar to those presented in Chapter 4. This strategy essentially works because, by successfully reformulating the STPU into a dispatchable form, the compiler proved that the dispatcher could schedule the rest of the plan almost as before, while giving up control over the uncontrollable durations, without causing a failure. Morris proved that doing this correctly only require inserting or modifying simple temporal constraints of the system and possibly adding *conditional constraints* to the problem [9]. A conditional constraint provides a simple temporal constraint that the dispatcher must enforce in scheduling a particular event in order to avoid squeezing an uncontrollable duration, unless some specified uncontrollable event has already executed. For example, we might constrain an event Y to happen at least 20 time units after X, unless Z has already happened, in which case the inequality does not matter because the execution of Z removes the need for conservatism in the scheduling of Y. The dispatcher adds reasoning about these constraints to the handling of execution windows and event orderings it already does, which is sufficient to manage the uncertainty correctly. Section 5.4 expands these ideas and describes the necessary algorithms. Drake adapts the algorithms designed for STPUs to consider the impact of discrete choices by representing families of related STPUs with the labeled data structures developed in this work and augmenting

the prior work to function on this representation.

5.2 Defining Plans with Uncertainty

In this section we define uncontrollable extensions for both TPNs and DTPs, both of Drake's basic representations. These uncontrollable varieties are denoted TPNUs and DTPUs, respectively. Since some of our most important innovations stem from the representation Drake uses to reason over the plans, we also present the necessary modifications to labeled distance graphs.

A TPNU is a TPN where some of the activities are marked as uncontrollable[1]To perform constraint reasoning, we transform the TPNU into a Disjunctive Temporal Problem with Uncertainty, following Venable and Smith [20]. The deterministic DTP definition is augmented with uncontrollable events and edges. Prior literature often use alternate terminology, referring to durations as requirement or contingent links.

Definition 5.3 (Disjunctive Temporal Problem with Uncertainty [20]) A DTPU is a tuple $\langle V_c, V_u, R_d, R_u, C\rangle$, where V_c and V_u are the controllable and uncontrollable events, respectively. R_c and R_u are the controllable and uncontrollable edges and C is the finite disjunctive constraints of the edges. □

We view both these formats as a means for specifying a family of related component STPUs. As before, the objective of this chapter is to develop the techniques required to dynamically dispatch either of these types of problems, determining the execution times and making choices regarding which of the component STPUs to execute on the fly.

A crucial step of this work for the deterministic case was the development of the labeled distance graph to represent the family of STPs in Chapter 3. Similarly, we represent uncontrollable problems with a similar data structure. Before we can do so, we formally define conditional constraints, which the new representation must include.

[1]We assume that discrete choices are always controllable. See Effinger et al. for TPNs with uncontrollable discrete choices [6].

Definition 5.4 (Conditional Constraint) A conditional constraint of the form $\langle t, B\rangle$ on directed edge (C, A) specifies that either B must execute before C or else $A - t \leq C$. □

Morris demonstrated that the addition of conditional constraints, which were first formulated as wait constraints, into the compiled form is sufficient to dispatch dynamically controllable STPUs. The compilation process for a STPU terminates with a distance graph that may include some conditional constraints. The key innovation of this section is to provide a constraint storage mechanism for uncertain constraints and conditional constraints, defined below as *Conditional Labeled Distance Graphs with Uncertainty*. First, the definition divides the events into controllable and uncontrollable ones. Second, it provides an annotation for weights of the graph, representing that the durations are either controllable or uncontrollable. Note that while the weights are annotated with their controllability, the domination function does not consider these annotations, because Drake only needs the tightest bounds known of either type. Finally, the new structure creates labeled value sets for conditional constraints. The conditional constraints can be stored in labeled value sets because they are distances in the graph and only the tightest known ones need to be kept. Drake maintains a separate labeled value set for each triple of events, indicating the start, end, and conditional events of the constraint.

Definition 5.5 (Conditional Labeled Distance Graph with Uncertainty) A labeled weighted distance graph G is a tuple $\langle V_c, V_u, W, C\rangle$. V is a list of vertices partitioned into controllable events V_c and uncontrollable ones V_u. W is a group of labeled value sets for the weights and the marking of the controllability of the edge $f((a, b), (a', b')) \leftarrow (a < a')$, where a is the weight and b is either C for controllable or U for uncontrollable. This group of value sets represents the weight function that maps vertex pairs and an environment to a weight and a controllability annotation: $V \times V \times \mathcal{E} \rightarrow \mathbb{R} \times \{U, C\}$ for any vertex pair $(i, j) \in V \times V$ and environment $e \in \mathcal{E}$. The set of edges E is those pairs where $w(i, j) \neq \infty \in W$ for some environment. All the labeled value sets for weights are initialized with the pair $((\infty, C), \{\})$. C is a

group of conditional constraints mapping triples $(i, j, k) \in V \times V \times V$ of events into labeled value sets. The first two indicate the direction of the inequality, and the third what the edge is conditional on. The conditional constraints are initialized with no elements. □

Converting from an input TPNU or DTPU to a conditional labeled distance graph with uncertainty is almost identical to the method for deterministic plans, given in Section 3.2. Essentially, the events and constraints are directly mapped between the representations, where an environment summarizes whether the constraint is part of a choice (see Example 3.6). The only difference for uncontrollable plans is that some events and constraints are annotated as uncontrollable in the conditional labeled distance graph. An event is considered uncontrollable if any uncontrollable duration, with any environment, ends at that event.

In the controllable problems, we developed the activity data structure to help the dispatcher reason about when activities begin, whether events are waiting for an activity to complete, and whether the dispatcher is committed and should begin an activity. This separate representation also provides the dispatcher with a central repository to search for the activities, regardless of whether the simple temporal constraints representing the duration was pruned away at compile-time. Since uncontrollable durations are similar in spirit to activities, and these same considerations apply, it is convenient to treat all uncontrollable durations as activities, where we augment the *act* data structure from 4.5, with a *controllable*? field, containing a Boolean value that indicates whether the activity is controllable. This mechanism simplifies our code by avoiding unnecessary duplication of steps.

With these data structures defined, our task is relatively simple. We can adapt prior algorithms for STPUs to handle choices by storing the plan in the conditional labeled distance graph with uncertainty and replacing all the operations from those algorithms with their labeled equivalents. These modifications are described in more detail in the next sections.

5.3 Compiling Plans with Uncertainty

Morris showed that a STPU can be reformulated into a dispatchable form through a polynomial time algorithm that transforms the input plan, replacing uncontrollable durations with controllable durations and conditional constraints that prevent squeezing of the uncontrollable durations at run-time [9]. This process is completed by repeatedly modifying certain pre-defined small sub-graph structures, thereby propagating the effects of the uncontrollability throughout the constraint graph. Stedl developed an efficient technique for achieving the same result by re-ordering the propagations and by modifying the rules for altering the graph [14]. This section presents the application of Stedl's method to our compact representation for families of STPUs, conditional labeled distance graphs with uncertainty. We do not change the core algorithm from Stedl's work, except to replace the operations with their labeled equivalents and by adding steps to record conflicts specifying infeasible component problems. Therefore, we give an overview of the strategy and some algorithms here, but other details are left to Appendix A, where we have duplicated, with permission, the chapter of Stedl's thesis that includes the derivations of the propagation rules.

Stedl's compilation algorithm for STPUs functions in two basic steps. The first step ignores the uncertainty and compiles the problem as if it were an STP. This compilation provides a dispatchable form of the problem that forms the base of the second step. Before concluding the first step, the algorithm checks for *pseudo-controllability*, ensuring that the constraints of the STPU do not imply a tightening of the activity durations. The second step modifies the graph to ensure that the execution windows of uncontrollable events are not tightened at run-time. It does this by applying a set of *back-propagation rules,* designed to update an STPU to maintain dispatchability if, for some reason, the dispatcher needs to change a constraint. Specifically, it takes the uncontrollable durations, which were treated as controllable in the first step, and propagates all changes needed in the dispatchable form to handle the fact that the durations are actually uncontrollable. Although Morris's iterative approach was the first proven to have polynomial run-time, Stedl's incremental strategy structures

the computations in a more efficient way, which is why we adopt it for this work. The back-propagation process either yields a new dispatchable form that handles the uncertainty, or an inconsistency is found. The back-propagation phase is when conditional constraints might be added to the problem.

Stedl's top level fast dynamic controllability algorithm for STPUs works to compile families of STPUs, with only accommodations for storing the edge weights in labeled value sets. The pseudo-code is shown in Algorithm 5.1. The method proceeds in two main phases: (1) compile the problem, treating every constraint as controllable, and test for pseudo-controllability, and then (2) propagate the effects of the uncontrollable durations. It takes as input a conditional labeled distance graph with uncertainty, formed from an input DTPU or TPNU, and either compiles it to dispatchable form or finds it infeasible.

Algorithm 5.1 Compilation algorithm for Temporal Plan Networks with Uncertainty

```
 1: procedure CompileTPNU(V_c, V_u, W, S, C)
 2:     W, S ← Labeled-APSP(V, W)
 3:     if (S == ∅) ∨ ¬PseudoControllable?(V, W, S) then        ▷ Alg. 5.2
 4:         return null
 5:     end if
 6:     W ← FilterSTN(V, W)                                      ▷ Alg. 3.6

 7:     for v ∈ V_u do                                  ▷ propagate uncontrollable
 8:         if ¬BackPropagateInit(V, W, S, C, v) then            ▷ Appendix A.
 9:             return null
10:         end if
11:     end for

12:     if ¬PseudoControllable?(V, W, S) then                    ▷ Alg. 5.2
13:         return null
14:     end if
15:     W ← FilterSTN(V, W)                                      ▷ Alg. 3.6

16:     return W, S
17: end procedure
```

We use the following example to illustrate the compilation algorithm.

Example 5.6 Consider the drive activity from the rover example, while making the

drive an uncontrollable duration. We also add a requirement that the rover warm up the science package, but not more than 10 minutes before the drive ends, to avoid wasting power. This fragment of the plan is depicted in Figure 5-1a. Event A starts the drive, event B ends the drive, and C issues the command to warm up the science package. Controllable edges are denoted with open arrows and uncontrollable constraints are drawn with filled arrows. The event on the end of the drive, B, is uncontrollable, denoted by the square node. □

The first phase of the algorithm produces a dispatchable network for the fully-controllable version of the STPU and tests it for pseudo-controllability. A STPU is pseudo-controllable if the implicit constraints of the network do not prohibit any values from the uncertain durations. This condition is necessary but not sufficient for dynamic controllability because the executive cannot select which value any the uncontrollable duration receives, and the executive must allow any possible value. To test pseudo-controllability, Line 2 runs LabeledAPSP on the graph to explicitly reveal all the implicit constraints of the problem. If APSP finds a negative cycle, then the problem is inconsistent, as before, because re-introducing the uncertainty makes the problem strictly harder and is certainly inconsistent. Then Line 3 actually performs the pseudo-controllability check on the compact representation of all the STPUs, invalidating any component STPUs that contain restricted uncontrollable durations. Assuming that at least some of the component STPUs are still feasible, the first phase concludes by filtering the network of redundant edges on Line 6, producing a minimal dispatchable form of the STPUs that passes pseudo-controllability, as needed for the next step. The filtering step must follow the pseudo-controllability check because the reason for running the APSP algorithm is to expose all the constraints and filtering removes them, potentially hiding the evidence that the problem is not pseudo-controllable.

The result of running the labeled APSP algorithm on Example 5.6 is shown in Figure 5-1b. The input graph has few edges, so only one new edge, (A, C) is created. No negative cycles are found, so the algorithm continues. The uncontrollable durations are not squeezed by any new edges, so the problem is pseudo-controllable. After

testing for pseudo-controllability, the graph is pruned, and edge (A, C) is pruned, because it is dominated by the other two. A controllable edge may be dominated by an uncontrollable edge. In contrast, we cannot remove any uncontrollable edges from the graph at this stage, because they need to be propagated in the next step.

The second phase of compilation considers the effects of propagating timing information at run-time and ensures that the dispatcher never tightens the uncontrollable durations incorrectly. Stedl developed a set of *back-propagation rules* that specify a method for updating a network to maintain dispatchability when a constraint changes. Line 8 performs the primary reasoning step. This step changes the uncontrollable durations, which the first phase treated as controllable, back into uncontrollable ones and recursively propagate the necessary timing changes throughout the network. We do not review the back-propagation rules in detail, but leave their derivation to Appendix A. The back-propagation rules detect inconsistencies caused by the revisions to the network that are performed to avoid execution window tightening, creating conflicts for violating component STPUs, as usual. Afterward, Line 12 re-checks that pseudo-controllability was not violated during back-propagation. Finally, the compact, dispatchable representation of the consistent STPUs is pruned of redundant edges on Line 15, in order to make dispatching more efficient. At this stage, uncontrollable durations may be pruned, as any uncontrollable durations the dispatcher needs are stored in the activity data structure.

Although this chapter does not review the back-propagation rules used during the second phase of compilation in detail, the following example demonstrates the derivation of a conditional constraints derived by a back-propagation rule.

Example 5.7 The conditional constraint is derived as follows. The drive might last between 30 and 70 minutes and the time between warming up the science package and the drive ending must not be more than 10 minutes. The drive might end at any time during the allowable uncertain duration. Therefore, scheduling the warming event 40 minutes into the drive might cause an error as the executive cannot guarantee that the drive will end less than 50 minutes into the execution. Once the rover is 60 minutes into the drive the executive may conclude that it can warm up the science package

because the drive is guaranteed to end in less than 10 minutes, so all the requirements will be met, regardless of the possible remaining outcomes. However, if the drive does end at some earlier time, the executive need not be so conservative; once the drive is over, this constraint is satisfied and the science package warm-up may be scheduled at any time, subject to other constraints of the plan. A conditional constraint is created to encode this knowledge: after the drive it started, the executive must not start warming up the science package until either 60 minutes have passed or the drive completes, whichever is first, denoted $\langle -60, B \rangle$. The dashed line in Figure 5-1c depicts this new constraint. Previous literature used wait constraints, which invert the duration, encoding the same constraint as $\langle 60, B \rangle$ instead [10]. Stedl changed the notation into conditional constraints to make it consistent with the semantics of the distance graph, which we adopt here. □

Using the back-propagation rules on labeled values simply involves placing the union of the two input environments on the new value. In the above example, say the 70 minute constraint had an empty environment, $\{\}$, and the 10 minute constraint had environment $\{x = 1\}$. Then the resulting conditional constraint, $\langle -60, B \rangle$, would have their union, $\{x = 1\}$, as its environment. The back-propagation rules are applied to edges with the function LabeledBinaryOp, presented in Algorithm 3.4, which performs this environment operation.

Since pseudo-controllability is an important part of STPU compilation, we provide Algorithm 5.2, which demonstrates the minor adaptation necessary for the labeled representation. The essential idea is that in all valid STPUs, the uncontrollable durations must not be replaced by tighter durations. Line 2 loops over every uncontrollable duration in the original specification, searching for violations. The inner loop on Line 3 considers every edge between the same events as the uncontrollable one under investigation. Any smaller weights indicate a tightening that might violate pseudo-controllability. The tighter weight means that the dispatcher cannot simultaneously satisfy the environment of the uncontrollable duration and the lower weight edge. Therefore, Line 4 computes the union of those two environments and invalidates it by creating a conflict. Finally, Line 10 returns that the problem is controllable if

at least some complete environments remain valid.

Algorithm 5.2 Algorithm for testing pseudo-controllability on Drake's compact representation and updating valid set of environments.

```
1:  procedure PseudoControllable?(V,W,S)
2:      for every uncontrollable edge (w, e) from event i to j do
3:          for (w_ij, e_ij) ∈ W_ij do
4:              if (w_ij < w) ∧ (e subsumes e_ij) then
5:                  AddConflicts(S, e_ij)                          ▷ Section 3.3
6:                  RemoveFromAllEnv(e_ij)
7:              end if
8:          end for
9:      end for

10:     if EnvironmentsRemain?(S) then   ▷ return true if some STPUs are left
11:         return false
12:     else
13:         return true
14:     end if
15: end procedure
```

Example 5.8 Figure 5-2 shows two small graph segments we use to illustrate the pseudo-controllability algorithm. First, Figure 5-2a has exactly one uncontrollable edge. The algorithm would look for any violating edges, immediately finding the only other edge from event A to B, which has a smaller weight. Therefore, the environments of the edge weight $\{x = 1\}$ is now a conflict. However, the DTPU is still valid if there are other complete environments.

Figure 5-2b shows an interesting similar case, with the same edge weights, suggesting that some solutions should be marked as invalid. However, the uncontrollable edge and the tighter bound do not co-occur in any environments because they differ in their assignment to the variable x. The algorithm determines this incompatibility by computing that $\{x = 2\}$ does not subsume $\{x = 1\}$. Therefore, the algorithm draws no new conclusions from this fragment. □

This completes our description of the compilation algorithm for problems with temporal uncertainty. We have explained the top level dispatching algorithms, adapting Stedl's technique for STPU compilation to consider the effects of choices through a

labeling scheme. The compilation algorithm is built from the same APSP algorithm and filtering algorithm used for the controllable case and two new steps: testing for pseudo-controllability and back-propagating the uncontrollable constraints. More detail on Stedl's controllability algorithm are provided in Appendix A. Given the techniques and data structures developed for the controllable case, extending Drake's compilation process is a relatively simple process.

5.4 Dispatching Plans with Uncertainty

Morris et al. proved that dynamically dispatching a compiled STPU requires two simple modifications to the STP dispatcher that Drake copies [9]. First, some events are not directly controllable and the system must wait for them to complete; this is exactly the same as waiting for the end events of activities to complete and is essentially already handled by the pseudo-code for Drake. Second, the dispatcher needs to respect the conditional constraints at dispatch time. These additional constraints alter when certain events can be executed and therefore require modifications the event selection algorithm. We also slightly modify the activity selection algorithm so it does not attempt to control the durations of uncontrollable activities. The top level routines and propagation techniques are identical to those presented in Chapter 4, except that they must call the two modified functions presented here.

The most important update to the dispatcher is to modify the event selection routine to respect conditional constraints, meaning that to execute an event with a conditional constraint, either the conditional event has executed or the difference constraint is satisfied. In the labeled version, as before, the executive may execute an event at a certain time if it can invalidate all the environments of violated constraints without removing all possible complete environments. The conditional constraints are now just another source of violated constraints the executive must search for. Algorithm 5.3 handles this extra consideration. Note that it is otherwise identical to Algorithm 4.3. Lines 20-28 loop over all the triples including this event. Note that conditional constraints are always negative and point out from the events the

Algorithm 5.3 Determine if an event is executable.

```
1: procedure EventExecutableU?(V, W, V_exec, S, B, C, i, t)
2:     e_violated ← {}
3:     for (a_j, e_j) ∈ B_i^u do                          ▷ Test upper bounds
4:         if a_j < t then
5:             e_violated ← e_violated ∪ e_j
6:         end if
7:     end for

8:     for (a_j, e_j) ∈ B_i^l do                          ▷ Test lower bounds
9:         if a_j > t then
10:            e_violated ← e_violated ∪ e_j
11:        end if
12:    end for

13:    for j ∈ V \ V_exec do                              ▷ Test activation
14:        for (a_k, e_k) ∈ W_ij do
15:            if a_k < 0 then
16:                e_violated ← e_violated ∪ e_k
17:            end if
18:        end for
19:    end for

20:    for j ∈ V do                                       ▷ Test conditional events
21:        for k ∈ V \ V_exec do
22:            for (a_m, e_m) ∈ C_i,j,k do
23:                if (j ∉ V_exec) ∧ (a_m < Time(j) − t) then
24:                    e_violated ← e_violated ∪ e_m
25:                end if
26:            end for
27:        end for
28:    end for

29:    if ConflictsPossible?(e_violated) then             ▷ Test for remaining solutions
30:        AddConflicts(e_violated)
31:        return true
32:    else
33:        return false
34:    end if
35: end procedure
```

constraint. Therefore, the outer loop is over the end of the edges and the middle loop is over non-executed events. This algorithm only needs to find violated constraints, so we need not test any constraints where the conditional event is executed and satisfies the constraints by definition, so the middle loop only searches over non-executed events. Otherwise the inequality constraint is tested, ensuring that the other event has actually been executed and if so, that the difference constraint is met. The operator Time(j) refers to the time of execution of event j. If either of these tests fail, the environment of the conditional constraint is added to those that must be discarded and the algorithm proceeds as before.

Example 5.9 Consider executing the labeled conditional distance graph depicted in Figure 5-3. Assume that event A was executed at $t = 0$. At the current time, $t = 10$, event B has not executed yet. If the dispatcher considers executing event C, it finds the conditional constraint. The constraint would be violated if Drake schedules event C at $t = 10$ because event B has not executed, nor have 60 minutes elapsed since event A executed. Therefore, the executive must create a conflict from the conditional constraint's environment, $\{x = 1\}$, in order to execute event C at $t = 10$. □

The modification to the activity selection algorithm is simple. If the activity the function commits to is not controllable, it calls BeginActivityU on Line 16, which does not attempt to set a duration for an uncontrollable duration. Therefore, it also skips the computation of duration of the activity. Otherwise, the function is unchanged. We create activities for the uncontrollable durations because the two concepts are semantically related, and it provides a convenient way to re-use the code that delays the execution of events until real-world activities allow it. Also, it ensures that pruning cannot remove the uncontrollable edges and therefore makes the dispatcher unaware of the uncontrollable duration that is ongoing. As before, our mechanism for waiting is simplistic, requiring that only one uncontrollable duration ends at each event. We can work around this restriction by either noting which activities must complete for a given event to execute or by having events constrained with a $[0, 0]$ edge, requiring that they happen simultaneously.

At the top level of the code, it is sufficient to remove the end event from the waiting list when uncontrollable duration completes, because our dispatcher executes events as soon as they are executable. The compiler guaranteed that the end event of any uncontrollable duration is executable at any possible outcome of that duration, so we can rely on the existing procedures to execute the end event on the first time step after the duration completes.

Algorithm 5.4 A function to begin activities starting with a given event.

```
1: procedure BeginActivitiesU(V, V_exec, W, S, B, i, t, Act)
2:     V_waiting ← ∅

3:     for acts.t.(act.start = i) ∈ Act do
4:         if EnvironmentValid?(S, act.e) then
5:             CommitToEnv(S, act.e)                          ▷ Commit to activity

6:             if act.controllable? then                      ▷ select a duration
7:                 t_exec ← ∞

8:                 for (a, e) ∈ B^l_act.end do                ▷ Search bounds
9:                     if EnvironmentValid?(e) then           ▷ Sec. 3.3
10:                        t_exec ← min(t_exec, a)
11:                    end if
12:                end for

13:                t_exec ← max(t_exec − t, act.e)

14:                BeginActivity(act, t_exec, act.end)        ▷ Start the controllable
   activity
15:            else
16:                BeginActivityU(act.end)     ▷ Start the uncontrollable activity
17:            end if
18:            V_waiting ← V_waiting ∪ act.end
19:        end if
20:    end for
21:    return S, V_waiting
22: end procedure
```

STP dispatchers and STPU dispatchers are relatively similar, except for the addition of conditional constraints and the need to wait for uncontrollable durations to complete. Therefore, updating Drake only requires a similar modification. The pro-

cess for selecting events and activities is identical, as is the strategy for committing to choices through the creation of conflicts.

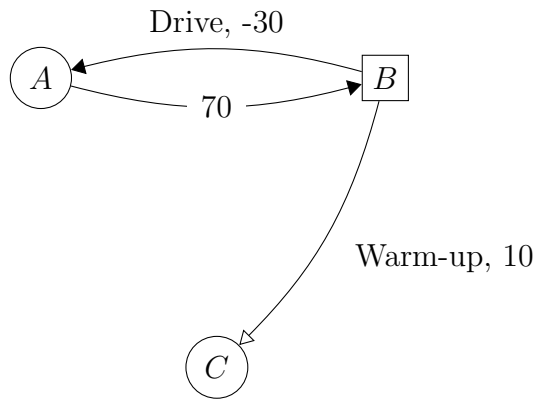
5.5 Conclusions

In developing Drake's technique for compiling and dispatching families of related STPUs, we outlined and adapted Stedl's compilation method for STPUs. We provided algorithms that first compile a deterministic version of the plan, then test that the problem is pseudo-controllable. Then the consequences of the uncontrollable durations are propagated throughout the plan, updating other constraints and possibly creating conditional constraints. We also updated the dispatching routines for controllable networks to respect these conditional constraints, while preserving the framework for compiling and dispatching described in Chapters 3 and 4.

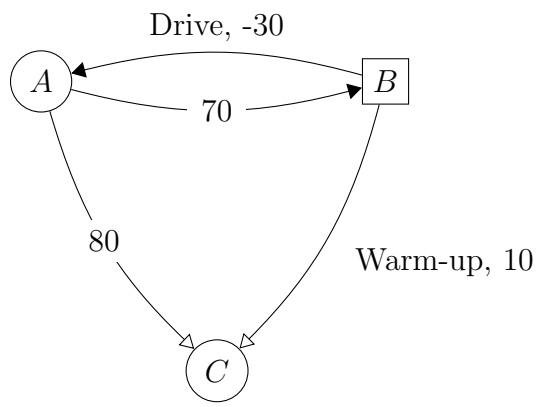
This section completes the presentation of technical innovations. Previous chapters have developed labeled value sets as an efficient and simple technique for applying non-disjunctive temporal reasoning to disjunctive problems. This chapter provided an interesting case study, because adapting the existing STPU algorithms only required the basic techniques already developed for Drake's deterministic algorithms. Labeled value sets are versatile enough to provide the backbone of the representation for the new conditional constraints and readily accept the new operations necessary to propagate uncontrollable durations. The ATMS was intended as a framework for supporting general problem solving engines, and we see some of this generality, specialized to weighted graphs.

Figure 5-1: An example of an uncertain duration and a conditional constraint that results. Depicts Examples 5.6 and 5.7.

(a) Input uncertain problem



(b) APSP Form of Fragment



(c) Resulting conditional constraint

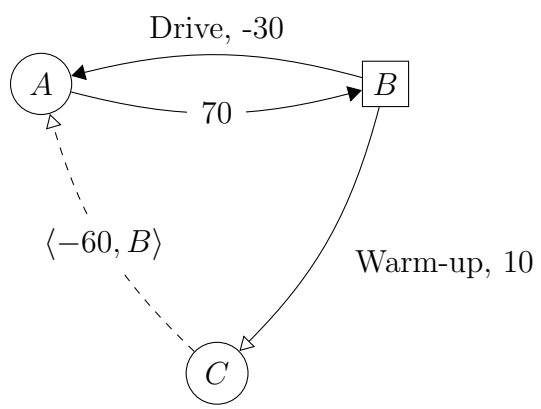
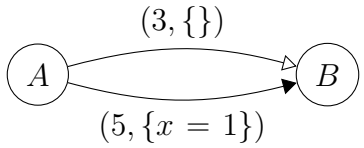


Figure 5-2: Graph fragments demonstrating pseudo-controllability.

(a) A tighter edge invalidating $\{x = 1\}$



(b) A tighter edge with no effect

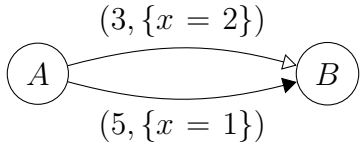
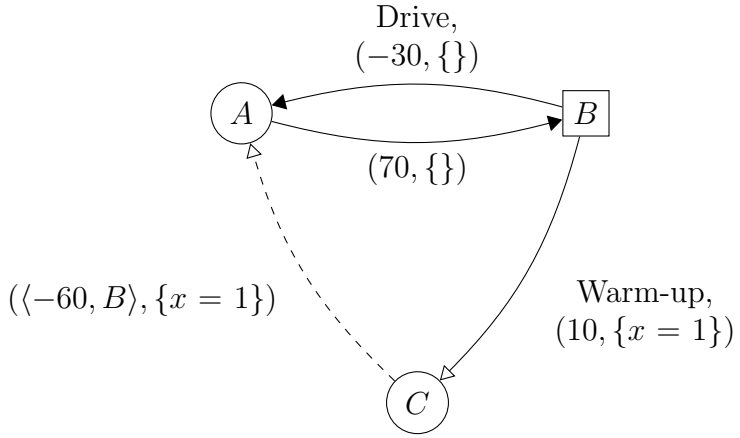


Figure 5-3: A fragment of a labeled conditional distance graph with uncertainty.



Chapter 6

Experimental Results

This chapter presents an experimental validation of Drake's compilation and dispatch algorithms on randomly generated, structured problems. First, we develop a suite of random structured DTPs and TPNs. Then we compile and dispatch the suites of problems twice, once with Drake and once by explicitly enumerating all the component STNs, following techniques developed in Tsamardinos's work [19]. Finally, we compare the compiled size of the problems, the compilation time, and the execution latency. We find that, in general, Drake's performance on TPNs, DTPs, TPNUs, and DTPUs are remarkably similar, regardless of the differences in structure or the presence of uncontrollable durations. The data shows that Drake's labeled distance graph representation, which we developed as a compact form of the component STP(U)s, is compact in practice. We see a consistent decrease in the compiled size of the problems compared to Tsamardinos's explicit enumeration, up to around four orders of magnitude for the largest problems, containing over 10,000 consistent component STPs. Drake's compilation time is often faster than the explicit enumeration process of Tsamardinos, but sometimes takes longer by up to two orders of magnitude. Finally, Drake's execution latencies are typically slower than Tsamardinos's work, sometimes by several orders of magnitude for large problems, but still take less than a second for most moderately sized problems, with a few thousand component STPs. Overall, Drake's techniques successfully trade off compiled space for processing time.

6.1 Generating Random DTPs

To generate random structured Disjunctive Temporal Problems, we modify Stedl's random structured STP generator [14]. His generator provides relatively fine-grained control over the size of the resulting problems, and is we used to generate a large test suite of problems.

We give a brief overview of the generation algorithm. Stedl's generator first creates activities, where each activity is specified as a strictly ordered duration between two events. These activities are randomly assigned coordinates of a grid, with each event having a unique coordinate, where the start event of an activity is directly to the left of the end event. This coordinate grid is wider than it is tall, so that the activities give the feel of a time-line when drawn. Since the generated activities do not share events, the generator then adds more constraints to connect the activities. These extra constraints are added by randomly selecting an event, then selecting another event that is nearby on the coordinate grid and adding a simple temporal constraint, where the bounds are selected randomly from a range proportional to the distance between the two events. This scaling of constraints provides structure, and is the key feature of the technique, because some events are placed closely on the grid and constrained to occur at similar times, while others are widely separated and therefore remain loosely coupled. We add to this technique by generating disjunctive constraints in a similar fashion. For each disjunctive constraint, the algorithm selects an event to focus on and selects the desired number of constraints from the existing constraints near the selected event. These constraints then appear in the disjunctive clauses. If the generator selects a constraint that is already in a disjunctive clause, it creates another constraint for the disjunct, placed between the same events, with newly generated bounds.

This generator produces problems where the events are naturally understood as existing on a type of time-line, making them reminiscent of problems humans might create. It also provides flexibility over the size of the problems created. We tie together several of the size parameters to create two basic controls. The first control

scales the number of disjuncts per disjunctive constraint. The second control scales the number of disjunctive constraints in the problem, which determines several other parameters. The number of activities is the same as the number of disjuncts so the increased number of disjunctive constraints do not become cluttered. Events are only created for the activities, so the number of events is fixed at double the number of activities, as each activity gets independent start and end nodes. Finally, the number of non-activity constraints, added after the activities are created, is roughly three times the number of disjunctive clauses. This parameter, along with the scaling of the random temporal values and some other fine parameters of Stedl's algorithm, are chosen empirically so that most of the problems are consistent, and many of the component STPs are consistent.

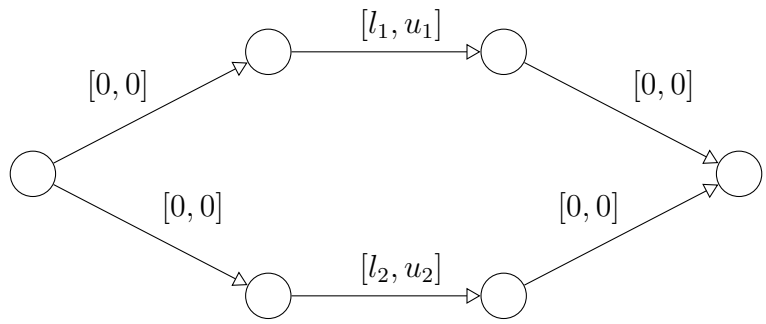
Our test suite varies those two controls, producing DTPs and DTPUs with up to thousands of component STPs. Specifically, we generated 100 consistent problems at each size, varying between each of 1-13 activities with 2 clauses per disjunctive constraint and each of 1-9 activities with 3 clauses per disjunctive constraint. We stopped increasing the activity size when the benchmarking time increased to several days per parameter increase. All the inconsistent problems were discarded.

When generating uncertain problems, each activity has a fifty percent chance of being marked as uncontrollable. Otherwise, the generation process is identical to the deterministic case.

6.2 Generating Random TPNs

Our TPN generator is based on the one presented by Effinger [5]. The algorithm creates a hierarchical plan by first creating a binary tree up to the desired depth, connected with $[0, 0]$ simple temporal constraints. Then each node is replaced with the TPN fragment shown in Figure 6-1. The two activities in the fragment are randomly generated durations where $0 \leq u_i \leq 10$ and $0 \leq l_i \leq u_i$. Each node in the binary tree and the left-most node of the TPN fragment is then converted into a choice with a probability of one half. Finally, the generator creates an end node for

Figure 6-1: This TPN fragment is the fundamental unit used by the random generation algorithm.



the TPN and connects the bottom of the tree, closing off the hierarchies and inserting new nodes appropriately.

As with the DTP generator, when generating uncertain problems, each activity has a fifty percent chance of being uncontrollable. Otherwise, the generation process is identical. We generated 100 consistent TPNs of depths one, two, and three. These problems are smaller than many of the DTPs, but increasing the depth to create larger TPNs also takes days to run. All inconsistent problems were discarded.

6.3 Numerical Results

To characterize the performance of Drake, we used it to compile and dispatch the test suites of controllable and uncontrollable TPNs and DTPs, created as explained above. Drake is implemented in Lisp, and all the evaluations were run in a single thread on a four core Intel i7 processor with 8 Gb of memory. Our implementation deviates slightly from the algorithms presented, however, the differences are largely superficial and there is a direct correspondence between operations performed by our implementation and our pseudo-code. The main difference is that the code does not use labeled value sets by name, instead, it uses multiple edges between any pair of events and places environments on those edges. However, the operations required to insert values and perform operations is essentially identical. Also worth noting is that some of the environment operations are memoized with a size limited hash table, which distinctly improves the computation times. As a point of reference for

comparison, we also implemented a compilation system that explicitly enumerates all the consistent STPs, as directed in Tsamardinos's work. We collected data on the compiled size of the problem, compilation time, and run-time latency, which we now present and discuss.

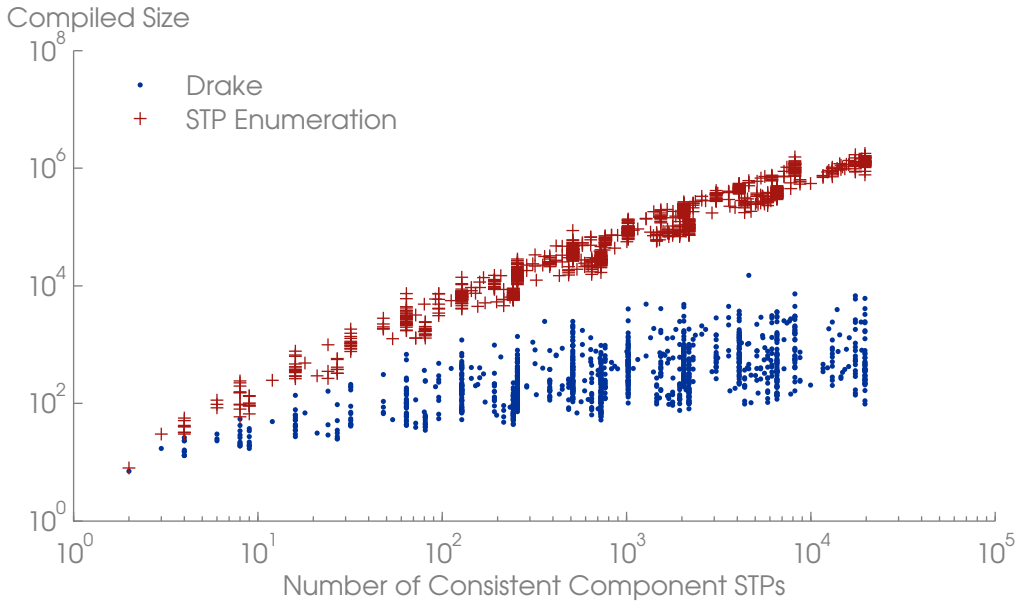
Throughout this section, our plots use the number of consistent STPs as the horizontal axis because seems to explain the variations in the data more clearly than the number of disjunctive constraints, which is the controllable independent variable of our generators. We believe it provides cleaner data trends because the fraction of feasible component problems varies dramatically. Therefore, a problem with many disjunctive constraints, but only a few feasible component problems might be easier to store than one with fewer disjunctive constraints, but all the components are feasible. Also, we developed Drake to avoid the costs that result from explicitly creating component STPs, hence it seems reasonable to study whether our method scales better than prior work against this variable. Our analysis of the data suggests that this variable provides the clearest indicator of the difficulty of the problem, in that it almost completely separates Drake from the explicit enumeration technique for the size metric we collected.

We begin with the size of the compiled representation, because it provides the clearest and most favorable results. The size is computed as a platform independent metric, counting all the important data structure elements stored by each representation. Drake's size metric counts the number of events, values in all the labeled value sets on the edges, and the overhead of the conflict database, measured as the number of conflicts and kernels. The STP enumeration metric counts the number of events and edges, summed over all the consistent, component STPs. Figure 6-2 presents scatter plots of the four types of problems, DTPs, DTPUs, TPNs, and TPNUs, on separate log-log scales, showing the compiled size versus the number of consistent STPs or STPUs in the problem. We selected a log-log scale to make the data visible over the large range of both axes.

The compiled size for all four types of problems show a similar trend: Drake's method provides a consistent and significant reduction in the size of the compiled

Figure 6-2: The compiled size of random problems as a function of the number of component STP(U)s.

(a) The compiled size of DTPs.



(b) The compiled size of DTPUs.

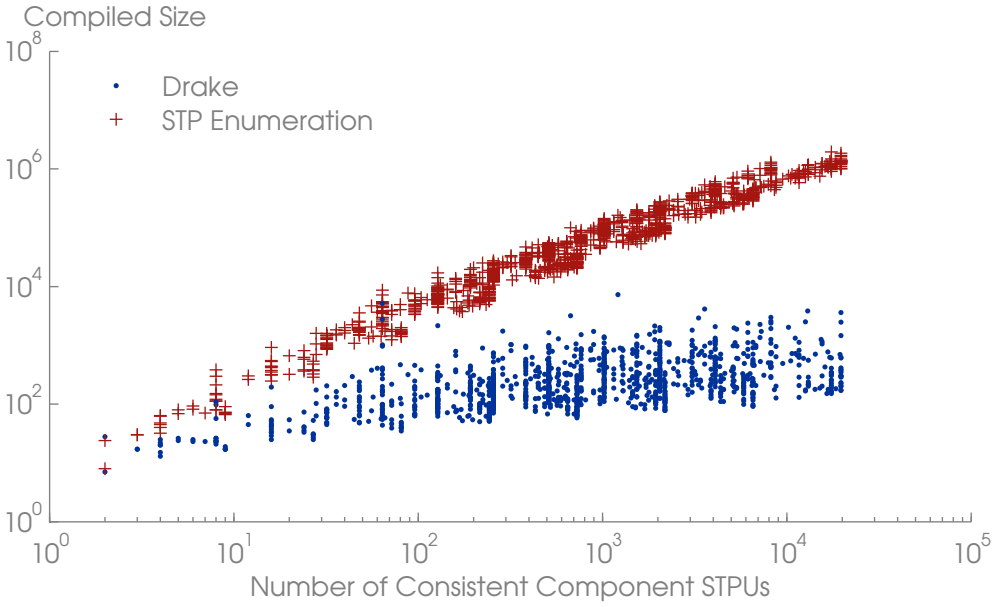
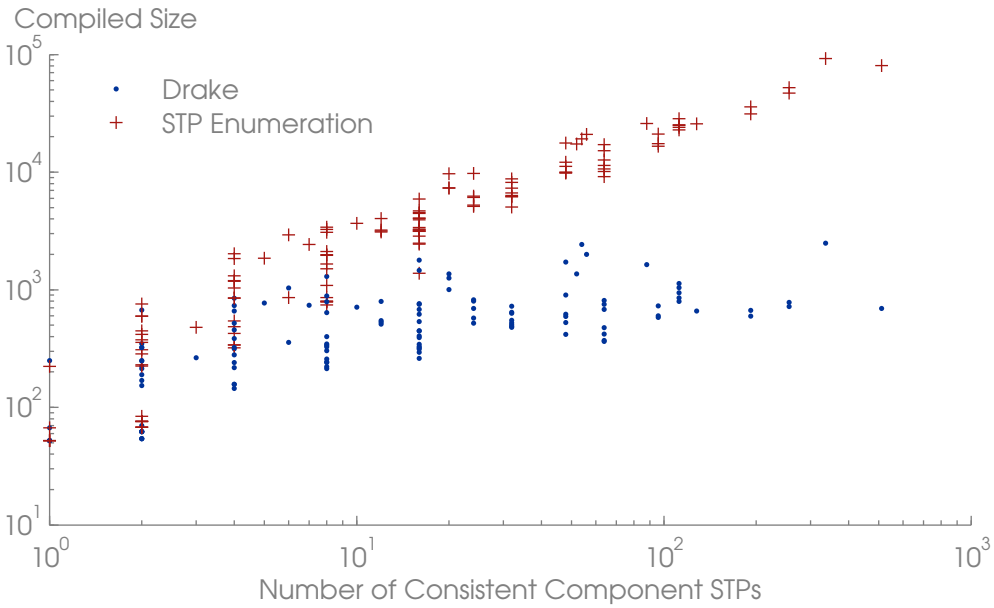
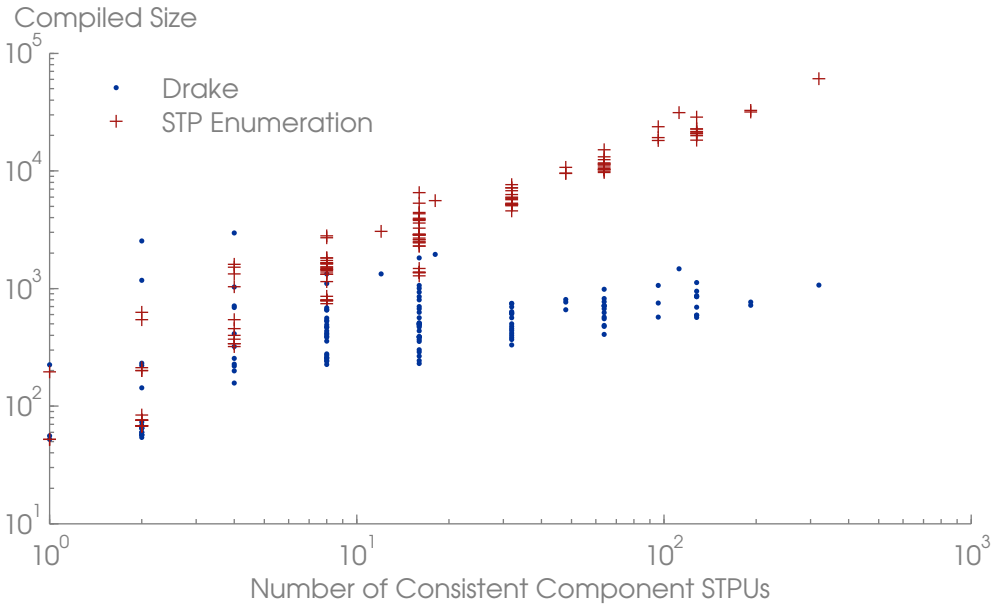


Figure 6-2: The compiled size of random problems as a function of the number of component STP(U)s (cont).

(c) The compiled size of TPNs.



(d) The compiled size of TPNUs.

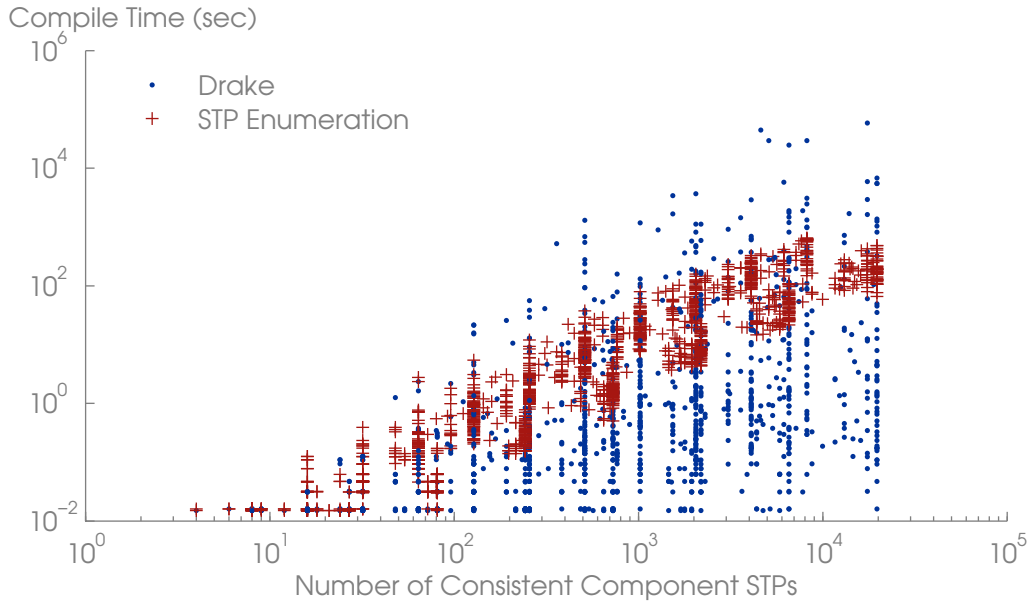


problem, typically ranging from one to four orders of magnitude in savings as the problem size increases. In fact, the qualitative shape of the graphs are identical across all four types, and the scales and slopes are relatively similar across all the graphs. Recall that our TPN generator creates smaller problems than the DTP generator and has less parameters, meaning that there are fewer data points and less variation in the number of component STPs within those data points. Even so, the TPN data scale essentially as the DTP data does. The DTP graphs show that varying the number of disjuncts per disjunctive clause does not change the trend, as the two sets of data are completely mixed. Instead, the number of consistent options is the only factor that matters. Furthermore, the presence of uncontrollable durations has little influence on the graph. These graphs clearly show that storing the component STPs or STPUs of a problem using labeled distance graphs reduces the number of events and edges as compared to the requirement for storing each component separately. Furthermore, there is little fixed overhead for storing STPUs. Instead, the cost is simply to store any additional conditional constraints required for dispatch, which are similar in form to the simple interval constraints. This result validates our primary claim of the compactness of Drake's representation in comparison to direct enumeration.

The second metric we collected is the time required to compile the problem to dispatchable form, measured in seconds. Drake was timed while it compiled the entire problem and the direct enumeration strategy was timed while it compiled only the consistent component problems. Although explicit enumeration was under-counted by only timing while it counts the consistent component problems, the fraction of consistent components was never vanishingly small and should not move the points qualitatively on a logarithmic scale. The plots are shown in Figure 6-3, shown on log-log plots as before. Again, the results are remarkably similar throughout the four types of problems. Directly enumerating the STPs is a very consistent process and it clearly costs polynomial time with respect to the number of consistent component problems, just as we expect. Drake's performance, on the other hand, varies considerably. For many DTP or DTPU problems, Drake either matches or dramatically outperforms Tsamardinos's strategy, but there is a small, yet noticeable

Figure 6-3: The compile time of random problems as a function of the number of component STP(U)s.

(a) The compile time of DTPs.



(b) The compile time of DTPUs.

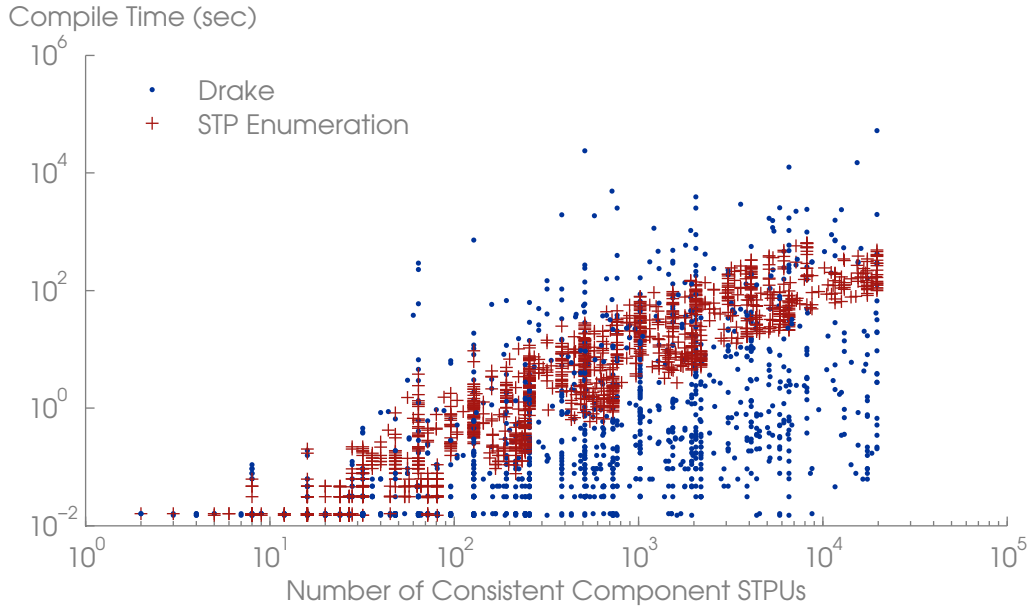
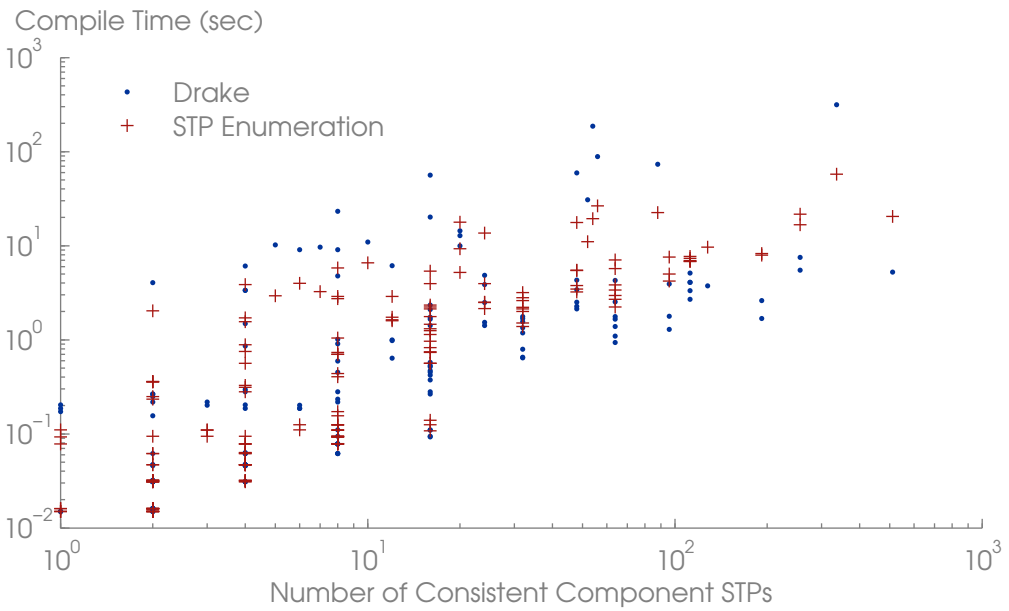
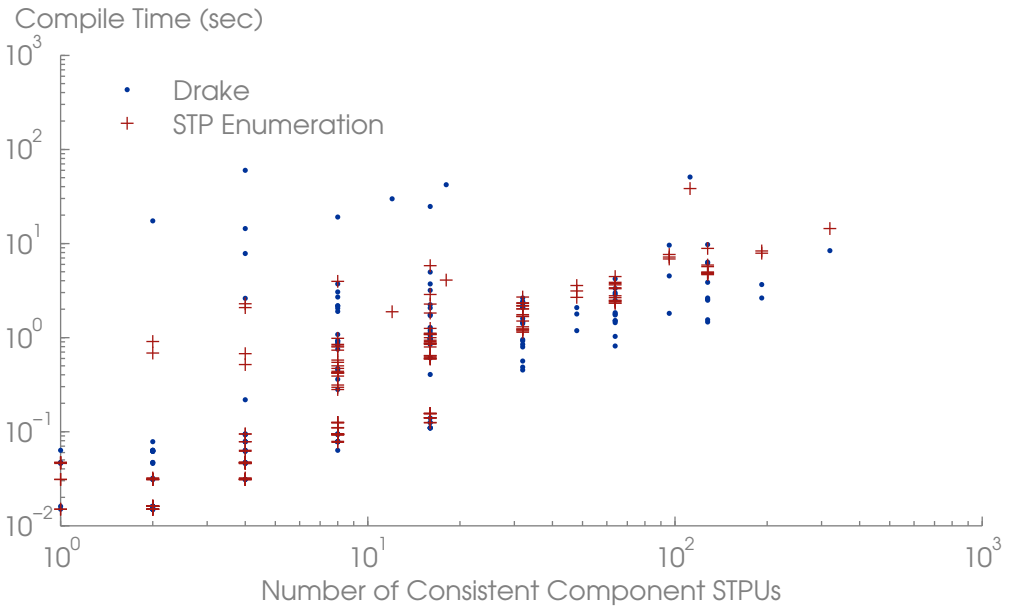


Figure 6-3: The compile time of random problems as a function of the number of component STP(U)s (cont).

(c) The compile time of TPNs.



(d) The compile time of TPNUs.



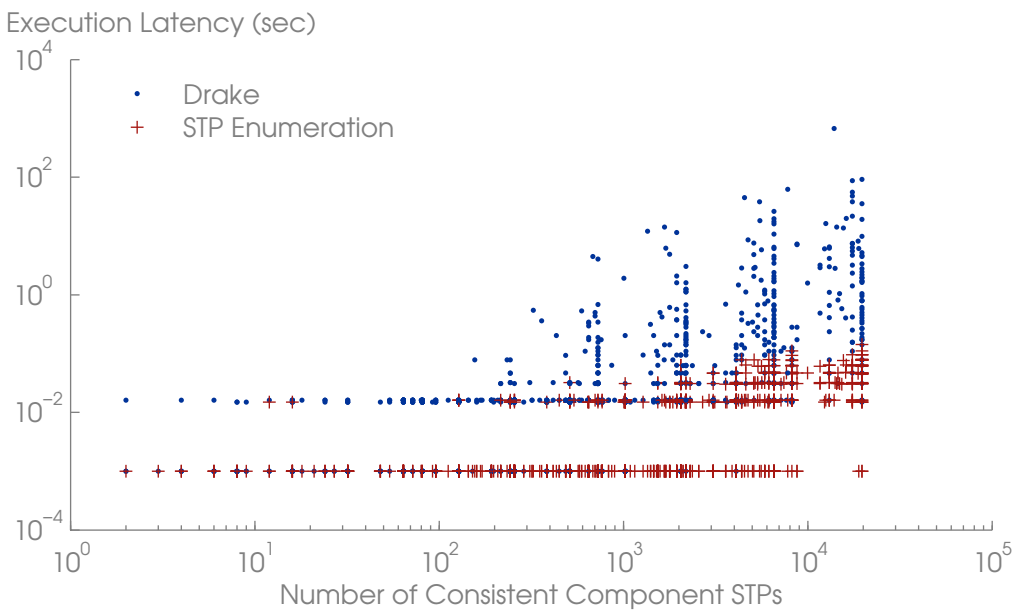
fraction where Drake's compilation time is an order of magnitude or two worse than Tsamardinos's strategy. We believe that the savings is representative of reduced redundancy in the computations during compilation for closely related problems, which is also something we hoped to see in the data. Unfortunately, in some problems, the computations involving the environments induce significant overhead, which we expect is correlated with component STPs that are relatively dissimilar. The TPN and TPNU graphs appear quite mixed and we cannot draw conclusions about one method outperforming another, but the results are consistent with the data seen in the DTPs and DTPUs with few component STP(U)s. Finally, we observed qualitatively, but cannot support numerically, that the filtering process is often the most expensive part. We believe this is because that algorithm searches over all pairs of values on intersecting edges, which is especially slow on the APSP form of the labeled distance graph.

The final metric we present is execution latency. To collect this data, Drake simulated running the plans once, of which we recorded the longest decision making period, during which Drake selected events to execute and performed propagations. The STP enumeration strategy was timed for the first dispatch step, which is not an upper bound on the execution time, but is representative because the number of STPs generally decreases during successive execution steps, and the work required to dispatch each one is relatively constant without environments to manage. The results are shown in Figure 6-4. The values at 10^{-3} were actually reported by Lisp's timing functions as zero, so we inflate them to place them on the logarithmic scale; the clear floor in the data at 10^{-2} is the minimum reported time.

Generally, Drake takes significantly more time to make decisions for large problems than Tsamardinos's approach, which we suspect is because of the extra labeled operations required at run-time. Although the increase in latency is sometimes two or three order of magnitude worse, the absolute speed of Drake's execution is generally not problematic. For small and medium sized problems, up to a few hundred component STPs, most execute with less than 0.1 seconds of latency. Even for the largest problems tested, many of the problems execute with less than a second of latency.

Figure 6-4: The execution latency of random problems as a function of the number of component STP(U)s.

(a) The execution latency of DTPs.



(b) The execution latency of DTPUs.

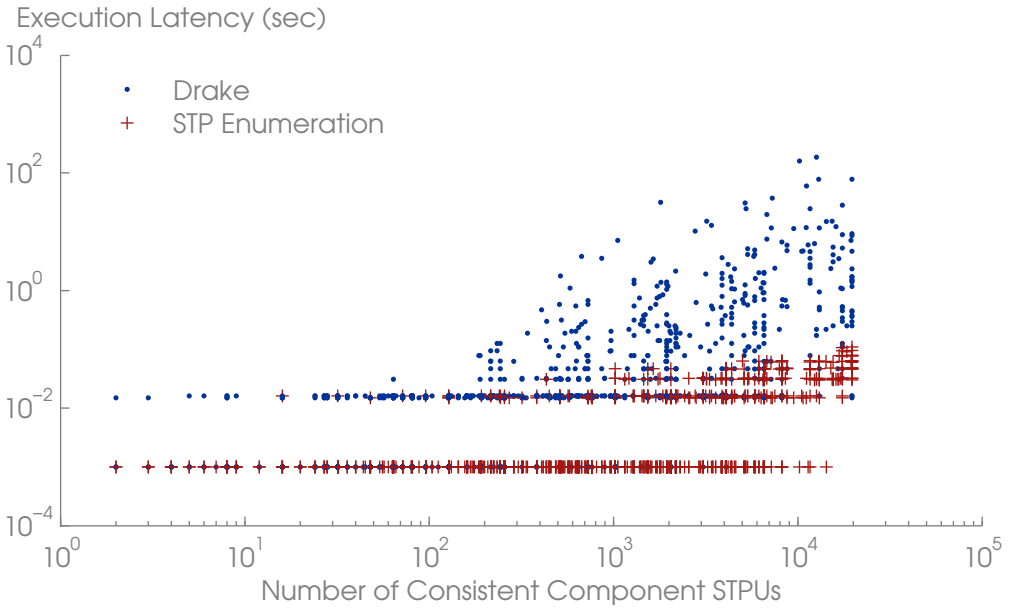
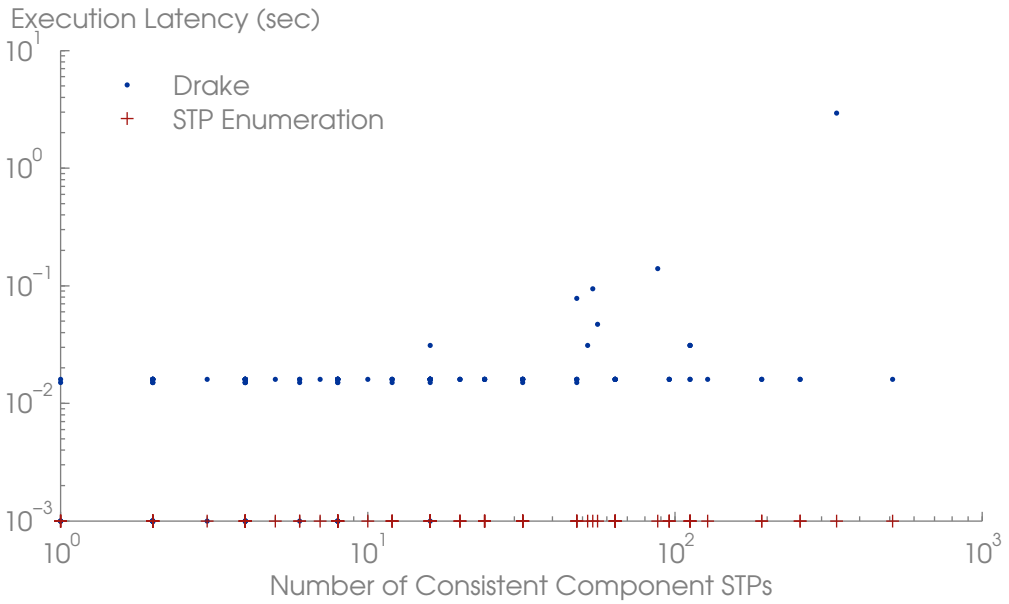
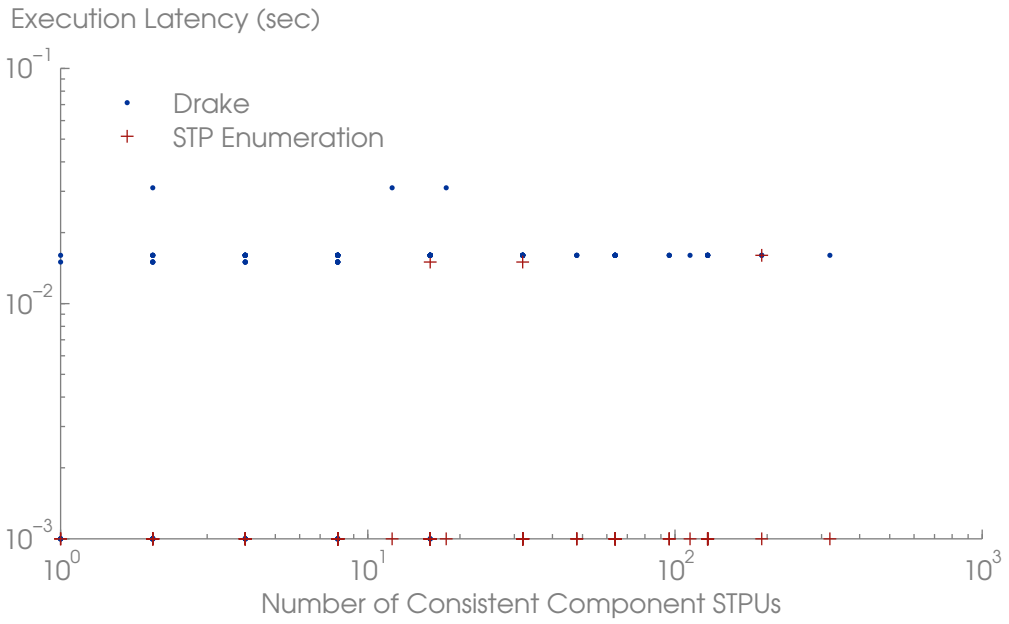


Figure 6-4: The execution latency of random problems as a function of the number of component STP(U)s (cont).

(c) The execution latency of TPNs.



(d) The execution latency of TPNUs.



Unfortunately, a few DTPs do suffer from latency in the tens of seconds for at least one reasoning step. Essentially all the TPNs execute with unmeasurable latency with explicit enumeration and the minimum measurable time for our timing functions with Drake, excepting one outlier, because the TPNs we tested are all small or medium sized. Similarly small latencies are visible in the small DTP and DTPU problems.

The overall conclusion we draw from these results is that Drake's compact encoding is indeed compact for several types of input problems, but also that it trades space for processing time in a manner that is typically favorable. However, we hesitate to draw more specific conclusions about the performance on any individual problem, because we cannot guarantee that the structured random problems we experimented with are representative of most real-world problems. The uniform results on two vastly different types of problems, including the time-line structure of the DTPs and the strict hierarchy of the TPNs, do lend credibility that these trends are relatively insensitive to some changes in the problem structure. However, future work is required to determine the distribution of compiled sizes, compile times, and run-time latencies a system could expect on real problem structures and constraints.

Chapter 7

Conclusions and Future Work

We presented Drake, a flexible executive for plans with choice. Drake is designed to take input plans with temporal flexibility and discrete choices, specifically DTPs or TPNs, potentially with uncontrollable durations, and decides the execution times and makes discrete decisions at run-time [4, 8]. Building upon prior work on the ATMS, Drake introduces a new compact encoding, called labeled distance graphs, to encode and efficiently reason over alternate plans [2]. This representation is especially useful because it requires relatively minor changes to non-disjunctive graph algorithms, in order to reason over the discrete choices. Drake's compilation algorithm successfully compresses the dispatchable solution by up to around four orders of magnitude relative to Tsamardinos's prior work, often reducing the compilation time, and typically introducing only a modest increase in execution latency. However, there are some cases where Drake performs poorly either at compile time or at dispatch time, relative to Tsamardinos's approach.

There are several possible avenues of research to improve Drake's performance. First, labeled distance graphs use a restricted variant of the labeling machinery developed for the ATMS, where each value in the labeled value sets are justified with a single environment, and values may exist in each set multiple times. In the original ATMS, each value would exist in the value set exactly once and is supported with a set of environments, which enumerate every partial assignment that would logically entail that value. Making this change might produce a more compact representation.

Managing sets of environments also introduces some design decisions about how to simplify them.

Example 7.1 Assume there is a discrete choice, x, with domain $\{1, 2, 3\}$. If a value is justified with environments $\{x = 1\}$, $\{x = 2\}$, and $\{x = 3\}$, the value actually holds universally because it is labeled with all possible options. Therefore, that value could be labeled with the single environment $\{\}$, simplifying the reasoning. □

The above example illustrates *hyper-resolution*, and is called "hyper," because it requires unifying more than two clauses simultaneously. A full ATMS labeling system with many disjuncts per disjunctive clause, thus requiring variables with large discrete domains, must efficiently perform hyper-resolution to keep the labels minimal and efficient.

Next, we observed during testing that filtering redundant edges from the labeled distance graph is often a computationally intensive step for Drake. We believe this step is especially expensive for Drake because the labeled values are represented in a single list, making it harder for the compiling algorithm to find useful domination tests.

Example 7.2 Take two edges that end at the same event, having labeled value sets $\{(2, \{x = 1\}), (4, \{x = 2\}), (6, \{x = 3\})\}$ and $\{(1, \{x = 1\}), (3, \{x = 2\}), (5, \{x = 3\})\}$. The filtering algorithm needs to search for dominated values because all these values are positive weights on edges ending at the same event. Labeled values cannot dominate one another if their environments differ on an assignment to any variable. Therefore, there are only three possible dominations here, matching values against the ones with identical environments. However, our algorithm searches over all pairs of labeled values, performing nine operations. In contrast, explicitly enumerating the component STPs performs this operation in minimal time, if there is only the one choice variable, x. Our filtering algorithm pays a penalty in run-time that is at least quadratic in the length of the value sets, compared to the time to filter a single STP with the same number of edges. The data suggests that for some of the examples we tested, this increase in cost is significant. □

Fortunately, in general, compiling an STP to dispatchable form by computing the APSP and then pruning redundant edges is known to be sub-optimal. Tsamardinos presented a fast technique, based on Johnson's algorithm, which avoids creating a complete graph through the APSP process before pruning it [17]. Instead, the propagation and pruning are interleaved to provide an efficient algorithm. This algorithm could be adapted to create a direct substitute for the labeled APSP algorithm we provide.

Finally, it should be possible to create hybrids between Drake's fully labeled approach and Tsamardinos's enumeration approach. Specifically, we could select a set of choices which are explicitly enumerated and others which are managed through a labeling approach. This should allow a continuous spectrum of performance between the space intensive but low execution latency solution provided by Tsamardinos and the compact but slower dispatching representation Drake uses. One difficulty we cannot provide firm guidance on, however, is how to appropriately select which choices to split on and which to label. However, we can suggest that it is probably desirable to to enumerate the choices that have the largest impact on the constraints of the problem and label the choices that induce less sweeping changes. This might make sense, because the labeled technique is designed to reduce the cost of representing related families of STPs, so we preferentially label the choices that create similar component STPs.

An alternative way to enhance Drake's feature set, rather than speed up the existing features, is to perform some type of reasoning about the utility of the choices available. Currently, activities, events, and choices are selected greedily. In real-world situations, where the user is likely to have some preference over the possible outcomes, Drake might be more useful if it could select executions with some notion of optimality.

Moving away from possible improvements to Drake, there are some broad lessons we can take from the development of Drake and the technique it uses. STP reasoning is largely made efficient by reformulating the STP questions into graph problems. We desired a system that could natively perform those same reasoning steps while

considering the impact of discrete choices. Therefore, we developed labeled distance graphs and specifically designed analogues to the APSP algorithm and a few other graph queries, such as dominance. Taking this work a step further, we could envision an entire graph package, with all the standard graph algorithms, but based on labeled value sets. We expect there are other uses in autonomous systems, such as path planning, or other fields, such as communications, where a labeled graph package could simply provide a compact encoding and an algorithmic strategy to interleave reasoning over choices with other existing algorithms.

In conclusion, Drake provides the first dynamic executive for TPNs and a development on prior DTP executives. It finds a new use for the representations underlying the prior work in ATMSs, compactly encoding solution sets for related families of STPs, without forcing us to derive completely new algorithms for temporal reasoning. This ability to dynamically make discrete choices from a compact representation will help robots to be more flexible and reactive in the future.

Bibliography

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, second edition, 2001.

[2] J. De Kleer. An assumption-based TMS. *Artificial intelligence*, 28(2):127–162, 1986.

[3] R. Dechter and R. Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 171(2-3):73–106, 2007.

[4] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[5] Robert Effinger. Optimal Temporal Planning at Reactive Time Scales via Dynamic Backtracking Branch and Bound. Master's thesis, Massachusetts Institute of Technology, 2006.

[6] Robert Effinger, Brian C. Williams, Gerard Kelly, and Michael Sheehy. Dynamic Controllability of Temporally-flexible Reactive Programs. In *Proceddings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 09)*, September 2009.

[7] D.J. Goldstone. Controlling inequality reasoning in a TMS-based analog diagnosis system. In *AAAI-91 Proceedings*, pages 512–517, 1991.

[8] P. Kim, B.C. Williams, and M. Abramson. Executing reactive, model-based programs through graph-based temporal planning. In *International Joint Conference on Artificial Intelligence*, volume 17, pages 487–493, 2001.

[9] P. Morris, N. Muscettola, and T. Vidal. Dynamic control of plans with temporal uncertainty. In *International Joint Conference on Artificial Intelligence*, volume 17, pages 494–502, 2001.

[10] N. Muscettola, P. Morris, and I. Tsamardinos. Reformulating temporal plans for efficient execution. In *Principles of Knowledge Representation and Reasoning-International Conference*, pages 444–452, 1998.

[11] A. Oddi and A. Cesta. Incremental forward checking for the disjunctive temporal problem. In *ECAI*, pages 108–112, 2000.

[12] Julie A. Shah and Brian C. Williams. Fast Dynamic Scheduling of Disjunctive Temporal Constraint Networks through Incremental Compilation. In *Proceedings of the International Conference on Automated Planning and Scheduling*, September 2008.

[13] I-hsiang Shu, Robert Effinger, and Brian C Williams. Enabling Fast Flexible Planning Through Incremental Temporal Reasoning with Conflict Extraction. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS 05)*, pages 252–261, 2005.

[14] John Stedl. Managing temporal uncertainty under limited communication: a formal model of tight and loose team coordination. Master's thesis, Massachusetts Institute of Technology, 2004.

[15] John Stedl and Brian C Williams. A Fast Incremental Dynamic Controllability Algorithm. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS 05)*, pages 69–75, 2005.

[16] K. Stergiou and M. Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence*, 120(1):81–117, 2000.

[17] I. Tsamardinos, N. Muscettola, and P. Morris. Fast transformation of temporal plans for efficient execution. In *Proceedings of the National Conference on Artificial Intelligence*, pages 254–261, 1998.

[18] I. Tsamardinos and M.E. Pollack. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence*, 151(1):43–89, 2003.

[19] I. Tsamardinos, M.E. Pollack, and P. Ganchev. Flexible dispatch of disjunctive plans. In *6th European Conference on Planning*, pages 417–422, 2001.

[20] K.B. Venable and N. Yorke-Smith. Disjunctive temporal planning with uncertainty. In *19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 1721–22. Citeseer, 2005.

[21] T. Vidal. A unified dynamic approach for dealing with temporal uncertainty and conditional planning. In *Fifth International Conference on Artificial Intelligence Planning Systems (AIPS-2000)*, pages 395–402, 2000.

[22] B.C. Williams and R.J. Ragno. Conflict-directed A* and its role in model-based embedded systems. *Discrete Applied Mathematics*, 155(12):1562–1595, 2007.

Appendix A

Reference on Fast Dynamic Controllability

The following material is the chapter of Stedl's thesis that develops the fast dynamic controllability algorithm for STPUs, and is reprinted with permission of the author [14]. We replicate this chapter in its entirety because Drake directly adapts this algorithm and specifically uses the back-propagation rules without modification or derivation. Therefore, we elected to provide this material for the completeness of the thesis, without trying to improve upon Stedl's explanations. We expect that the chatper should stand alone, given the background material provided in this thesis, but the entire text is available online.

4 Fast Dynamic Controllability Algorithm

4.1 Introduction

This chapter finishes the explanation of the Hierarchical Reformulation Algorithm introduced in Chapter 3, by describing how to reformulate each group plan into a dispatchable group plan. Specifically, this chapter introduces a novel efficient, centralized, dynamic controllability algorithm, called a *fast dynamic controllability algorithm* that transforms a plan constrained by an STNU into a dispatchable form. This chapter also describes a new edge filtering algorithm that transforms the dispatchable group plan into an efficiently dispatchable plan, called *a minimal dispatchable plan*. Together, the fast dynamic controllability algorithm and edge filtering algorithm perform *group plan reformulation*.

The goal of group plan reformulation is to enable the dispatcher to efficiently, dynamically, and consistently execute the group plan. The reformulation algorithms presented in this chapter are analogous to the reformulation algorithm, described in Chapter 2. Recall that in Chapter 2 we considered plans constrained by a STN; however, here we consider plans constrained by a STNU. We need to deal with uncertainty. Recall that in Chapter 3 we dealt with this uncertainty of the activities in the mission plan by decoupling the activities. This decoupling procedure enabled the activities in the mission plan to be executed independently. However, in this chapter we seek to precompile the temporal constraints of the group plan such that the agents can react to the uncertainty at execution time, in order to exploit the fact that the agents can communicate within the group plans. We seek to preserve some flexibility in the group plans so they can react to their situation at execution time, rather than simply preparing for the worst.

After the reformulation, the agents of the group must cooperate in order to execute the group plan. In the simplest approach, each group plan is executed using a leader-follower architecture. In this approach, a single leader is commissioned to make all scheduling decisions. The leader manages the execution process by sending commands to and receiving execution updates from the other agents in the group. All information passes through the leader. In Chapter 5, we present an alternative approach that distributes the execution process such that all the agents take part in the scheduling process. No matter which approach (leader-follower or distributed) is used, the fast dynamic controllability algorithm is still applicable.

This chapter builds on the concepts presented in Chapter 2 and Chapter 3. Specifically, the fast dynamic controllability algorithm expresses the temporal constraints of the group plan as a distance graph, then uses a set of local shortest path computations to reformulate the plan. The shortest path computations are a type of constraint propagation. Recall that constraint propagation is the process of deriving the feasible assignments on one set of variables given a set of constraints on different sets of variables. The fast dynamic controllability algorithm generalizes the strong controllability rules used in Chapter 3.

This chapter introduces one fundamentally new concept, the idea of a conditional constraint, which was originally introduced by [Morris 2001]. A conditional constraint (or wait constraint) is a ternary constraint (i.e. relates three timepoints) that is satisfied either by the passage of a minimum amount of time or through the notification of an event, which ever is sooner. It is similar to a lower bound simple temporal constraint, except that its enforcement is conditioned on the outcome of some other event. We use these types of constraints in everyday life. For example, consider a scenario where you plan on meeting a friend for lunch; however, you are running late. In such a scenario, you may call your friend and tell him you are running late. You would like to eat together, but, you do not want to be inconsiderate, so you ask your friend to wait for at least 20 minutes before ordering. However, if you get there before 20 minutes, there is no need to wait any longer. If your friend agrees, he has agreed on a conditional wait constraint. Your friend will wait for at least 20 minutes or until you arrive, which ever is sooner.

In this chapter, we show how to use these types of conditional wait constraints to preserve flexibility in partially controllable plans so that they can be dynamically executed. Beyond being desirable, the completeness of the fast dynamic controllability algorithm depends on using these conditional wait constraints.

The conditional wait constraint is a fundamental departure from the constraints we have been using, because they are ternary constraints (relate three timepoints), rather than binary constraints (relate two timepoints). Fortunately, [Morris 2001] showed that these conditional constraints can be propagated through the constraint network similar to simple temporal constraints. Furthermore, the introduction of conditional constraints only requires a small change to the STN dynamic dispatching algorithm, as presented in Chapter 2.

The dynamic controllability problem is solved by iteratively applying a set of local constraint propagation rules. Our fast algorithm builds on the basic structure of the dynamic controllability algorithm introduced by [Morris 2001]; however, it removes the need to perform repeated calls to an $O(N^3)$ All-Pairs Shortest-Path (APSP) algorithm.

The speed of the fast dynamic controllability algorithm is derived by two main innovations. First, our new dynamic controllability algorithm filters redundant constraints from the distance graph, up front, which reduces the number of propagations required. Second, we show that after performing a single APSP computation, the temporal constraints are placed in a *pseudo-dispatchable* form. Given this pseudo-dispatchable form, each constraint only needs to be resolved with the constraints that involve timepoints that occur earlier in the plan. Therefore, the constraints *back-propagate* through the distance graph. Applying these back-propagation rules allows the fast dynamic controllability algorithm to incrementally build up the reformulated distance graph, starting from constraints that relate timepoints that occur at the end of the plan, to constraints that relate timepoints that occur early in the plan.

The outline for this chapter is as follows. First we introduce some definitions and concepts related to dynamic controllability. Next, we review the dynamic controllably algorithm introduced by [Morris 2001]. Then we introduce our novel fast dynamic controllability algorithm. Finally, we introduced the new edge trimming algorithm. The empirical results for the new fast dynamic controllability algorithm are presented in Chapter 6.

4.2 Overview

This section reviews some key concepts and introduces several definitions regarding dynamic execution of plans that contain uncertainty. These definitions will be useful in subsequent sections.

A plan is dynamically controllable if there is a viable, dynamic execution strategy to schedule the timepoints in the plan. Recall that an execution strategy is viable if it generates a consistent schedule in all situations, and an execution strategy is dynamic if each scheduling decision is based only on the past.

The goal of dynamic controllability algorithm presented in this chapter is to compile the temporal constraints of the plan into a form such that a dispatcher can use to dynamically execute the plan. This reformulation enables the dispatcher to adapt to the plan's uncertainty at execution time.

Recall that dynamic execution is a scheduling process in which the timepoints of the plan are scheduled in real-time (timepoints are executed and scheduled simultaneously). In order to understand how to do this dynamic execution for plans that contain uncertainty, let's review the general job of the dispatcher and how to dynamically execute plans that do not contain any uncertainty (i.e. plans constrain by STNs rather than STNUs)

The dispatcher, whether applied to plans that contain uncertainty or not, is constantly making two related decisions: 1) what timepoint to execute next, and 2) when to schedule each timepoint. The reformulation algorithm compiles the temporal constraints of the plan in order to enable the dispatcher to make these decisions properly and quickly. This compilation is composed of two tasks: 1) it computes a set of enablement conditions for each timepoint, and 2) it exposes the set of implicit constraints inherent in the original explicit temporal constraints.

In Chapter 2 we presented two reformulation algorithms (a basic version and fast version) along with a compatible dispatching algorithm for plans constrained by an STN [Muscettola 1998a, Muscettola 1998b]. Recall that the basic reformulation algorithm first computes the All-Pair Shortest-Path (APSP) graph of the plan's distance graph, which exposes the implicit constraints, then trims the redundant (dominated) edges. The resulting graph is called the minimal dispatchable graph. Recall that in the fast version, the APSP computation and edge trimming are interleaved. For plans constrained by an

STN, the enablement condition is simply a list of timepoints that must be executed. For each timepoint, after the set of enablement timepoints have been executed, then that timepoint becomes enabled. The set of enablement timepoints for a timepoint X is computed by compiling all timepoints that are related to X by outgoing non-positive edges.

During execution, the dispatcher is free to select any timepoint for execution that is both enabled and alive. A timepoint is enabled if the all of the enablement timepoints have been executed and a timepoint is alive if the current time falls between the timepoints execution window. Every time the dispatcher executes a timepoint it performs two updates. First, it sends a set of enablement messages to all timepoints waiting on that timepoint's execution, and second it uses the constraints in the reformulated distance graph to update the execution windows of neighboring timepoints. [Muscettola 1998a] showed that upper bound updates are propagated via outgoing positive edges and lower bound updates are propagated via incoming non-positive edges.

The reformulation and dispatching algorithms need to be modified to support plans constrained by STNUs. The dispatcher only has partial control over the execution of the timepoints. A plan is only dynamically controllable if the plan does not further constraint the uncontrollable durations. Both the reformulation algorithm and the dispatcher must respect the timebounds of the contingent links. Recall that a contingent link specifies a lower and upper bound on an uncontrollable duration. If the temporal constraints of the plan imply strictly tighter bounds on the uncontrollable duration, then the uncontrollable duration is *squeezed*. Specifically, an uncontrollable duration is *squeezed* if its lower bound is increased or its upper bound is decreased, as illustrated in **Figure 4-1**. If the uncontrollable duration is squeezed, then there exists a situation where the outcome of the uncontrollable duration falls outside of the specified timebounds; therefore, consistency of the execution is dependent on the outcome of some uncertain event.

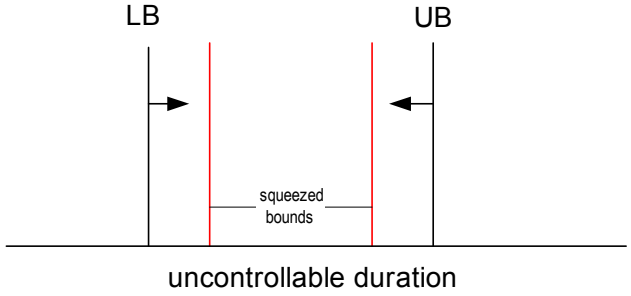


Figure 4-1 Each uncertain duration contains a lower and upper bounds as specified by the associated contingent link. The uncontrollable duration is squeezed if its lower bound is increased or its upper bound of the decreased

[Morris 2001] introduced the concept of *pseudo-controllability*, which provides a first check on the dynamic controllability of a plan. A plan in pseudo-controllable if it is temporally consistent and none of its uncontrollable durations are squeezed. The pseudo-controllability of a plan can be checked by computing the All-Pairs Shortest-Path graph (APSP-graph) of the plan's distance graph (ignoring the distinction between contingent and requirement edges). If the APSP-graph does not contain any negative cycles, and contingent edges remain unchanged in the APSP-graph, then the plan is pseudo-controllable. Therefore, if a plan is pseudo-controllable, then the contingent edges in the plan's distance graph are the shortest paths.

Example 4-1:

Consider the Distance Graph with Uncertainty (DGU) shown in **Figure 4-2**(a). The contingent edges represent the time bounds of the uncontrollable duration AB. The uncontrollable duration will last between [5,10] time units. The path ACB = 9 is shorter than the direct path AB = 10; therefore, the other constraints imply a tighter value on the upper bound of the uncontrollable duration. The APSP-graph shown in **Figure 4-2**(b) exposes this tightening. The uncontrollable duration is squeezed; therefore, the plan is not pseudo-controllable.

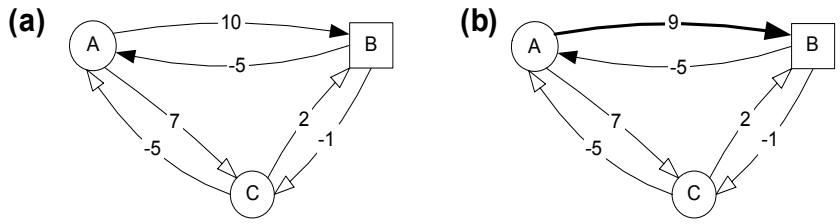


Figure 4-2 (a) The DGU with a uncontrollable duration between timepoints A and B (b) The APSP-graph exposes the temporal constraints imply a tighter upper bound on the uncontrollable duration; therefore, the uncontrollable duration is squeezed.

Example 4-2:

Consider the DGU shown in **Figure 4-3**(a). The contingent edges remain unchanged in the APSP-graph, shown in **Figure 4-3**(b). Furthermore, the plan is temporally consistent; therefore, the plan is pseudo-controllable.

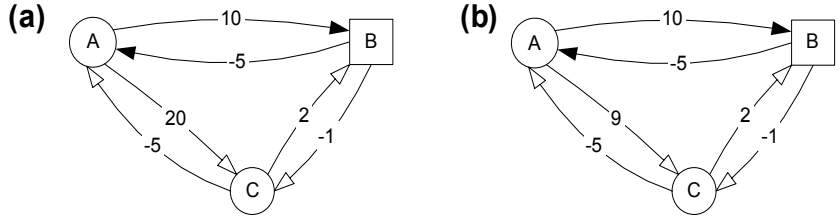


Figure 4-3 (a) A DGU with uncontrollable duration AB. (b) The APSP-graph does not further constraint the contingent edges.

Even if a plan is pseudo-controllable, the uncontrollable durations may be squeezed at execution time. When the dispatcher executes a timepoint, it effectively imposes a rigid constraint between the start of the plan and the timepoint being executed. If the dispatcher were to resolve this new constraint with the other constraints (by computing the APSP-graph) it may tighten a contingent edge, thus squeezing an uncontrollable duration.

Recall that the dispatcher does not need to recompute the APSP-graph every time it executes a timepoint. It updates the execution windows of the timepoints via a set of local propagations rather than updating the constraints of the plan. This is precisely the reason that the dispatcher is able to schedule the network in real-time. Therefore, when we talk about squeezing an uncontrollable duration during execution, it is more natural to express it in terms of the execution windows, rather than in terms of the temporal constraints of the plan.

Given a distance graph with uncertainty (DGU) with a positive upper bound contingent edge, AB, and corresponding contingent lower bound edge, BA, the execution window of the contingent timepoint B is squeezed if the execution window [x, y], resulting from propagation through edges AB and BA is tightened by any other propagation. Specifically, if the propagation through an incoming positive edge CB, where $C \neq A$, results in an upper bound, y', where $y' < y$, then the contingent execution window is upper bound squeezed. Similarly, if a propagation through some outgoing negative edge BC, where $C \neq A$, produces lower bound x', where $x' > x$, then the contingent execution window is lower bound squeezed. If the contingent execution window is squeezed during execution then the uncontrollable duration is also squeezed.

Example 4-3:

Consider the DGU show in Figure 4-4(a). The DGU is pseudo-controllable; however, it the dispatcher chooses execution time for B such that it squeezes the execution window of the contingent timepoint C. Timepoint A is the start of the plan and is executed at time = 0. After executing A, the dispatcher propagates the execution time through the plan. The execution window for C is [1,7] and the execution time for B is [5,10]. The dispatcher is free to choose any execution time for C between [1,7]. Figure 4-4(b) shows a case when the dispatcher chooses an execution time of 5 for B. After executing C, the dispatcher propagates the execution time of C through the edges CB and BC. These propagations result in an execution window for the contingent timepoint B of [8,10]. This squeezes the execution window of B. If the uncertain duration takes any time between 5 and 7 time units, then the execution is inconsistent. Note that if the dispatcher executed B at time 1 or 2, then the contingent execution window would not have been squeezed.

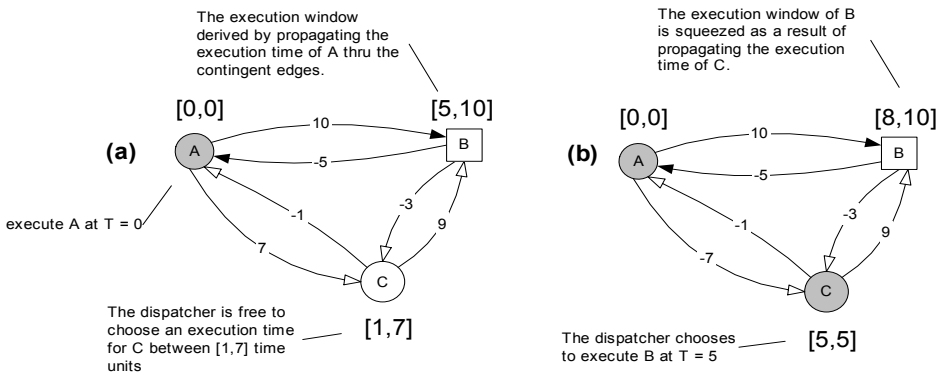


Figure 4-4 (a) The execution windows for the plan are shown after executing A at time = 0. (b) The execution window of the contingent timepoint B is squeezed from [5,10] to [8,10] after executing the timepoint C at time = 5.

The goal of the dynamic controllability algorithm is to add additional constraints to the plan in order to enable the dispatcher to consistently schedule the plan without squeezing consistent timepoints at execution time.

4.3 The Dynamic Controllability Algorithm

This section describes the dynamic controllability (DC) algorithm introduced by [Morris 2001]. The dynamic controllability algorithm transforms an STNU into a dispatchable graph. [Morris 2001] also showed that this algorithm is both sound and complete. If algorithm successfully reformulates the STNU, then the STNU is dynamically controllable; however, it the algorithm fails to reformulate the STNU, then the STNU is not dynamically controllable. In this section, we first present the overall structure of the DC algorithm, we will then describe the details of each step, and finally present the pseudo-code for the DC algorithm along with a brief analysis of the time complexity of the algorithm. In the next section, we present a new, faster, dynamic controllability algorithm, which is used in the Hierarchical Reformulation algorithm.

The dynamic controllability algorithm iteratively applies a set of *reductions* in order to prevent the dispatcher from squeezing the plan at execution time. These reductions are a set of rules that add (or tighten) the constraints to the plan. These reductions are similar to the strong controllability transformation rules presented in Chapter 3. The dynamic controllability algorithm uses a constraint processing loop that iterates between applying the reductions, propagating the effects of the reductions to the other constraints in the plan, and checking if the plan is pseudo-controllable. The DC algorithm loops until either 1) it determines that the plan is not dynamically controllable, by detecting an inconsistency or determining that the plan is not pseudo-controllable, or 2) it converges on a dispatchable graph.

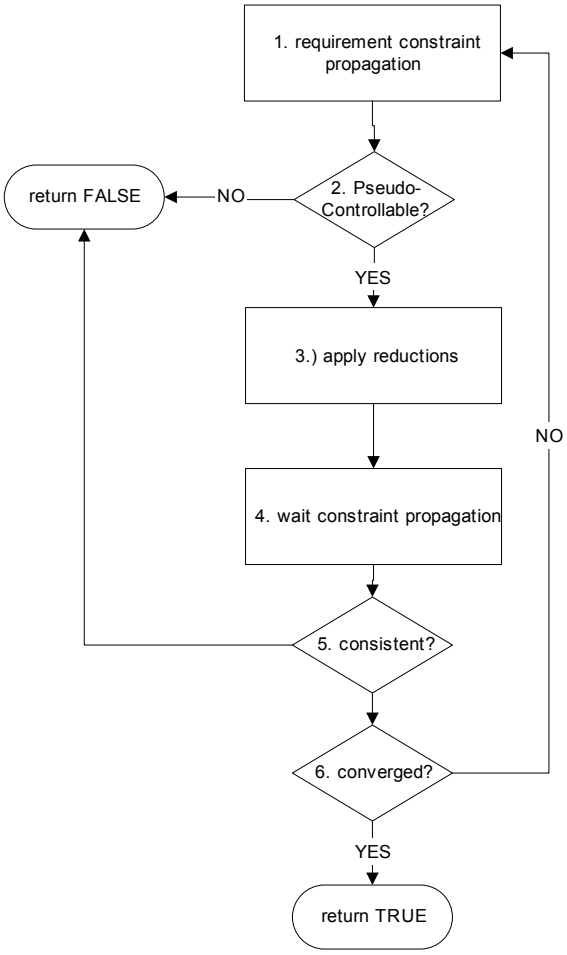


Figure 4-5 Basic Steps of Dynamic Controllability Algorithm

The constraint processing loop iterates between four basic steps: 1. requirement constraint propagation, 2. checking pseudo-controllability, 3. local constraint deduction, and, 4. wait constraint propagation, as shown in **Figure 4-5**. In Step 1, the algorithm resolves the simple temporal constraints by computing the All-Pairs Shortest-Path graph. Conceptually, after the first iteration, Step 1 propagates any change in the requirement constraints throughout the graph. In Step 2, the algorithm checks if any plan is pseudo-controllable. If the plan is inconsistent or any uncontrollable duration has been squeezed, the algorithm returns false. In Step 3, the algorithm applies a set of *reductions* in order to prevent an uncontrollable duration from being squeezed at execution time. The reductions may either modify the simple temporal constraints of the plan or modify the wait constraints of the plan. Finally, the algorithm propagates wait constraints through the plan.[3] If the algorithm determines any inconsistency during this wait constraint propagation, it returns false.

[3][Morris 2001] called the propagation of the wait constraints regression.

The algorithm loops through these four steps until either 1) the pseudo-controllability checking step fails, 2) the propagation of the wait constraints results in an inconsistency, or 3) the algorithm successfully goes through an iteration of the constraint processing loop without adding (or modifying) the constraints of the plan.

4.3.1 Triangular Reductions

This subsection describes a set of *reductions*, which add (or tighten) the temporal constraints of the plan, in order to prevent the dispatcher from squeezing the contingent execution windows at execution time. [Morris2001] derived the reductions in terms of a triangular STNU; however, here we will derive the reductions using the associated distance graph (DGU). It is more natural to use the distance graph because the dispatcher uses a distance graph to execute the plan.

The reductions are derived from a case analysis of a distance graph of a triangular STNU. The triangular STNU is shown in Figure 4-6(a) and the associated triangular DGU is shown in Figure 4-6(b). Later we show how the reductions derived for the triangular DGU are applied to distance graphs of arbitrary size. The triangular STNU consists of two executable timepoints, A, and C, and one contingent timepoint, B. It contains one contingent link AB $\in$ [x,y], corresponding to an uncontrollable duration, and two requirement links, AC $\in$ [p,q] and CB $\in$ [u,v]. We assume the STNU is pseudo-controllable and the distance graph is in an APSP form. Therefore, each edge in the distance graph corresponds to a shortest path distance.

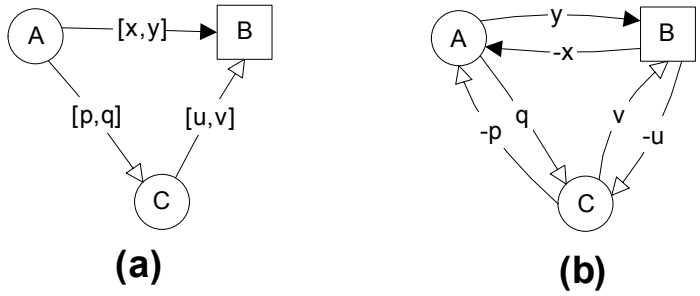


Figure 4-6 (a) The Triangular STNU (b) The associated triangular DGU.

The reductions are used to constrain the execution time of timepoint C, in order to prevent the propagations through CB and BC from squeezing the contingent execution window of B. Recall that the execution window of B can only be squeezed by incoming positive edges and lower bound squeezed by outgoing negative edges. We need to consider three cases. In the *precede* case, timepoint C must be executed before the contingent timepoint B. In the *follow* case, timepoint B must be executed after the contingent timepoint C, and in the *unordered* case, the execution order of B and C is undetermined. Recall that each execution order of a timepoint is determined by considering the negative edges in its DGU, as illustrated in **Figure 4-7**.

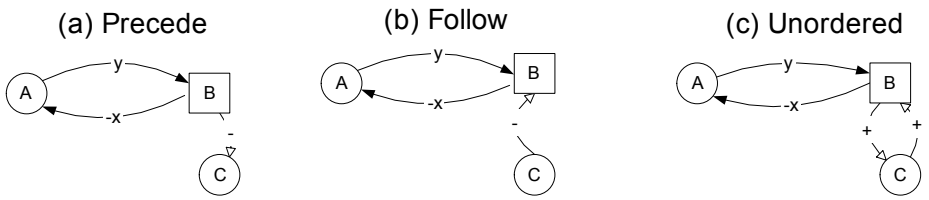


Figure 4-7 Temporal ordering relationships of a timepoint C with respect to a contingent Timepoint B.

Given a DGU, G, in an APSP-form, the order of execution of a timepoint, C, with respect to a contingent timepoint B, is as follows:

- A timepoint C must follow the contingent timepoint B, if there exists a negative edge BC in G.
- A timepoint C must precede the contingent timepoint B, if there exists a negative edge CB in G.
- The execution order of timepoint C is undetermined with respect to the contingent timepoint, B, if both edges BC or CB are non-negative.

Note that if both BC and CB are negative, then there exists a negative cycle and the DGU is inconsistent. In the triangular DGU shown in **Figure 4-6**, the ordering of C with respect to the contingent timepoint B is determined by the sign of u and v.

Follow Case: $u \geq 0$

If $u \geq 0$, then there exists a negative edge BC. In the precede case, timepoint C must be executed after the contingent timepoint B; therefore, the dispatcher is always privy to the execution time of contingent timepoint B when it makes the scheduling decision of C. Therefore, the dispatcher is able to adapt the schedule of C based on the execution time of B. In the follow case, the dispatcher uses edges BC and CB to update the execution window of timepoint C not B. Therefore, the execution of C will never squeeze the execution window of contingent timepoint B. As long as the STNU is pseudo-controllable, the dispatcher will be able to dynamically schedule B. The follow case requires no additional tightening of the constraints.

Example 4-4

Consider a scenario in which a student must meet with an advisor. The advisor's arrival time at the office is uncertain and will take between 5 to 10 minutes. Furthermore, the advisor requires at least 5 minutes to check his email before the meeting; however; the advisor is on a tight schedule so he does not want to wait in his office more than 10 minutes before the meeting. The advisor agrees to notify the student when he reaches his office. The student is willing to wait for up to 20 minutes. The student's plan is shown in **Figure 4-8**(a). The APSP-graph is shown in **Figure 4-8**(b). The APSP-graph is both consistent and the APSP-graph does not tighten the contingent edges; therefore, the plan

is pseudo-controllable. Furthermore, the timepoint C must follow timepoint B; therefore, the plan is dynamically controllable. **Figure 4-8**(c) shows the execution windows after executing timepoint A at T = 0. **Figure 4-8**(d) shows a situation where it takes the advisor 7 minutes to get to his office. This execution time is propagated to timepoint C. The new execution window for C is [12, 17]. The student can successfully execute the plan by getting to the office any time in this execution window.

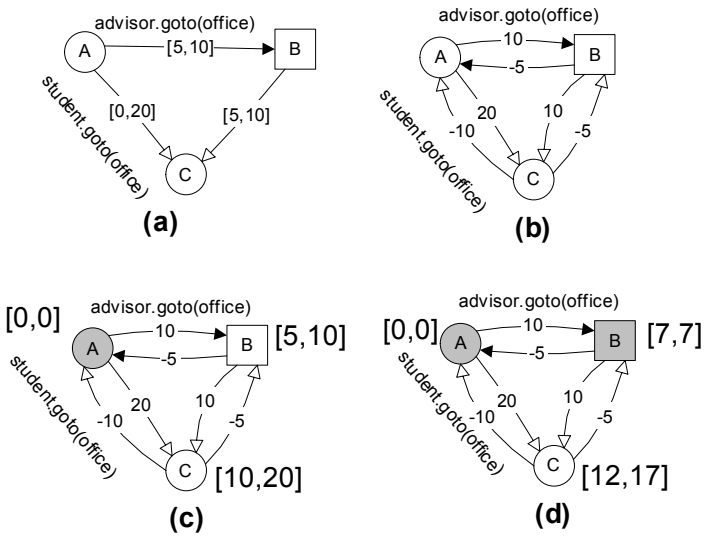


Figure 4-8 (a) In the student's plan, timepoint C must follow the contingent timepoint B. (b) The APSP-graph reveals that the plan is pseudo-controllable. (c) Timepoint A is executed at T = 0 and the execution windows are updated (d) Timepoint B is executed at T = 7, and the execution window for C is updated. In this situation, the student must get to the office some time between 12 and 17 minutes.

Precede Case: v < 0

If v < 0, then there exists a negative edge BC in the triangular distance graph; therefore, B must always be executed before C. In the precede case, the dispatcher will never know the execution of the contingent timepoint B when it needs to make the schedule timepoint C. This is exactly the situation addressed by strong controllability. The dispatcher is not able to adapt the schedule of C based on the execution time of the contingent timepoint B. The edge CB and BC are used to update the execution window of the contingent timepoint C. In order to be dynamically controllable, the algorithm must restrict the execution time of B. Specifically, in order to prevent the contingent execution window from being squeezed by propagations through CB and BC, we need to restrict the execution time of timepoint C with respect to A, by applying the appropriate strong controllability transformation rules. The reductions are simply the executable/contingent and the contingent/executable strong controllability transformation rules as derived in Section 3.5. However, instead of using the rules to compute a new transformed distance graph, as we did in the strong controllability algorithm, here the rules are used to directly

modify the distance graph. Specifically, the precede reductions tightens the edges AC and CA.

(Precede Reduction) *Given a triangular distance graph with uncertainty ABC (as shown in* **Figure 4-6***(b)), with v < 0, the edge AC is tightened to x-u, and the edge CA is tightened to v-y.*

As in the strong controllability case, the precede reduction effectively decouples the timepoint C from the contingent timepoint B. After applying the reduction, any propagation from timepoint C to timepoint B is redundant; therefore, the edges CA and AC can be removed from the distance graph. Any non-redundant information propagated through the edge CB and BC would only serve to squeeze the execution window of the contingent timepoint.

Note that the precede reductions are easily remembered, by first negating and transposing the contingent edges in the distance graph. Next, the shortest paths CBA and ABC are computed.

Example 4-5

Consider the STNU shown in **Figure 4-9**(a). The uncontrollable duration between timepoints A and B will take between 5 to 10 time units, and timepoint C must precede B by 1 to 8 time units. The APSP-graph is shown in **Figure 4-9**(b) is consistent and the contingent edges are not tightened; therefore, the STNU is pseudo-controllable. The edge BC is negative; therefore, C must precede B. In order to prevent the contingent execution window from being squeezed; we need to apply the precede reduction. The precede reduction tightens CA to -2 and AC to 4. **Figure 4-9**(c) shows the tightened distance graph. The edges BC and CB are not dominated. **Figure 4-9**(d) shows the distance graph after removing the dominated edges.

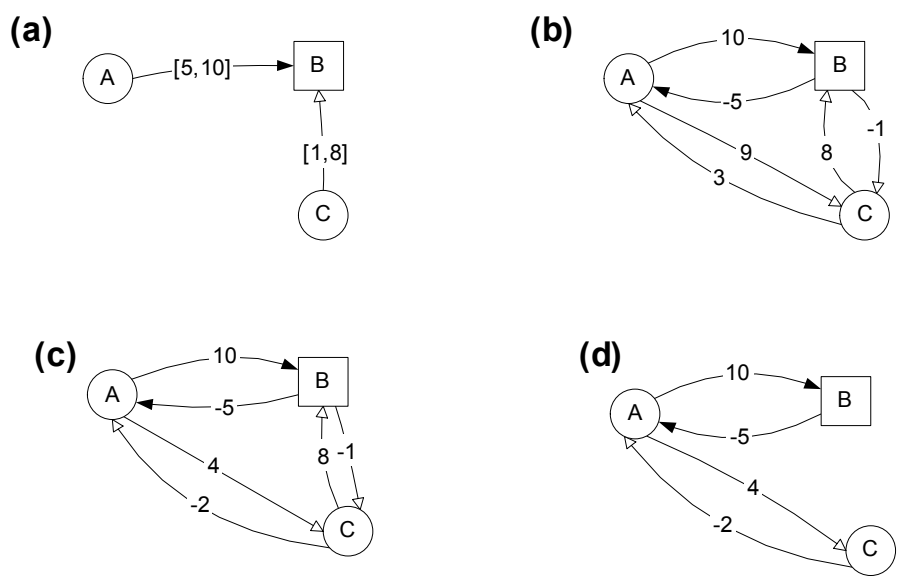


Figure 4-9 (a) The STNU where timepoint C must precede the contingent timepoint B. (b) The APSP-graph of the STNU. (c) The

resulting distance graph after applying the precede reduction. d(CA) + d(AB) = d(CB) and both AB and BC are positive; therefore, CB is dominated. Also, d(BA) + d(AC) = d(BC) and both BA and BC are negative, so BC is dominated. (d) The distance graph after CB and BC are removed.

Unordered Case: $v \geq 0$ and $u \leq 0$

In the unordered case, the edges BC and CB are both positive; therefore, the order of execution of B and C is not *a priori* determined. If C is executed first, then the edge CB is used to update the upper bound of the contingent timepoint B. However, if B is executed first, then the edge BC is used to update the upper bound of C. The simplest way to deal with the unordered case is to unconditionally constrain the execution time of B, in order to prevent the edge CB from squeezing C; this is accomplished by adding edge CA of v-y, as we did in the precede case. However, unconditionally constraining C may prevent the dispatcher from being able to react to the uncertain execution time of B, when B is executed first. Instead, we apply a softer constraint, called a *wait constraint,* which enables the dispatcher to adapt to the schedule of C when B is executed first, yet restricts the execution time of C in order to prevent B from being squeezed.

The wait constraint, written <B, t>, on edge AC specifies that the execution of C must wait for at least t time units after A executes or until B executes, which ever is sooner [Morris 2001]. In the previous example, B is called the *conditional timepoint* and t is the *wait duration.* Here we introduce a slightly different form of the wait constraint, called a *conditional constraint* or *conditional edge*, which encodes the same information as the wait constraint, except that it puts in a form similar to the edges in the distance graph. A conditional constraint is a directed edge that contains a distance expressing a temporal constraint similar to a requirement edge and a conditional timepoint similar to a wait constraint. The conditional constraint is the negative transpose of the wait constraint. A wait constraint <B,t> on an edge AC, corresponds to a conditional constraint of CA of <B,-t>. As in a requirement edge, the temporal distance of the conditional constraint requires that $T(C) - T(A) \leq -t$, which can be rewritten as $T(A) - T(C) \geq t$. If $t \geq 0$, then the conditional constraint encodes a lower bound temporal requirement (i.e. a wait condition) on C with respect to A. Similar to a wait constraint, this temporal requirement is only enforced until the conditional timepoint B is executed. After the conditional timepoint B executes, we say that the conditional constraint is relaxed. Thus, the conditional constraint CA of <B,-t> specifies that C must wait for at least t time units after A executes or until B executes, which ever is sooner.

In the unordered case, we apply a conditional unordered reduction, as defined below, which introduces a conditional constraint to the plan.

(Conditional Unordered Reduction) *Given a triangular distance graph with uncertainty (as shown in Figure 4-6(b)), where $v \geq 0$ and $u < 0$, apply a conditional constraint CA of <B, v-y>.*

After applying the conditional unordered reduction, if B executes first (follow case), then the conditional constraint is relaxed (i.e. the temporal requirement imposed by the conditional constraint no longer needs to be satisfied) and the dispatcher can react to the execution time of B. However, if C is executed first (precede case), then the temporal requirement of the conditional constraint ensures that the propagation from C will not squeeze the execution window of B.

Example 4-6

Here we revisit the student-advisor meeting problem with a slightly different temporal constraints. The advisor's arrival time is still uncertain. It will take him between 5 and 15 minutes to get to his office, and the student is willing to wait for up to 20 minutes before getting to the office. Both the student and the advisor will only wait a small amount of time in the office. The student will not wait more than 5 minutes after getting to the office, and the advisor, being more impatient, will wait no more than 1 minute. Furthermore, the student and advisor agree to call one another when they reach the office.

The student's plan for this scenario is shown in **Figure 4-10**(a). The APSP-distance graph is shown in **Figure 4-10**(a). The APSP-graph is consistent and the contingent edges are not squeezed; therefore, the plan is pseudo-controllable.

Consider the student's execution strategy. If the student gets to the office anytime before 10 minutes, he runs the risk that he will be waiting more than 5 minutes before the advisor arrives. For example, if the student only waits for 6 minutes, the student will be waiting for more than 5 minutes in a situation where the advisor arrives any time between 11 and 15 minutes. However, if the student unconditionally waits for 10 minutes, the advisor may be waiting around for more than 1 minute after he arrives. For example, if the student waits for 10 minutes and the advisor arrives in 7 minutes, then the advisor will be waiting around for 3 minutes. There is no unconditional strategy for successfully scheduling the arrival time of the student. Applying the conditional unordered reduction encodes a conditional execution strategy. The conditional constraint CA <-10,B> (dashed line), shown in **Figure 4-10**(c), specifies that the student must wait for at least 10 minutes or until the advisor arrives. This enables the student to successfully execute the plan.

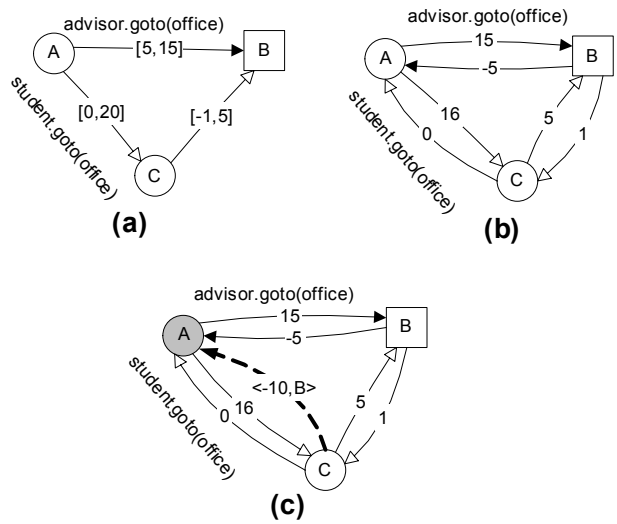


Figure 4-10 (a) The student's plan where the execution order of B and C is unordered (b) The APSP-graph of the student's plan (c) The distance graph after applying the conditional unordered reduction

In order to formally incorporate the conditional constraints with the Distance Graph with Uncertainty (DGU), we introduce a Conditional Distance Graph with Uncertainty (CDGU). The CDGU is a DGU that contains a set of conditional constraints. The dispatcher uses the information contained in the CDGU while executing the plan. The distance graph shown in **Figure 4-10**(c) is a CDGU.

Definition (CDGU): *A CDGU is a 5-tuple* $<N_{ctg},N_{exe},E_{req},E_{ctg},E_{cond}>$ *where* N_{ctg} *is a set of contingent timepoints,* N_{exe} *is a set of executable timepoints,* E_{req} *is a set of requirement edges,* E_{ctg} *is a set contingent edges, and* E_{cond} *is a set of conditional edges.*

There is one important case when a conditional constraint is actually unconditional. In this case the conditional constraint is converted into a requirement edge. Specifically, a conditional constraint is unconditional if the lower bound of the uncontrollable duration associated with a conditional timepoint is greater than the wait duration specified by the conditional constraint. In this case, the conditional timepoint will never be executed before the wait period is completed; therefore, the dispatcher must always wait the full duration, as specified by the conditional constraint. The *unconditional unordered reduction* specifies when a conditional constraint is converted into a requirement edge.

(Unconditional Unordered Reduction) Given a CDGU with conditional constraint CA of <B,-t>, and an uncontrollable duration AB $\in$ [x,y] associated with the conditional timepoint B, if x > t, then the conditional constraint CA is converted into a requirement CA with distance –x.

Note that the unconditional unordered reduction always applies when the temporal distance of the conditional constraint is positive. Therefore, after applying the unconditional unordered reduction, only negative conditional constraints remain.

Example 4-7:

Consider the CDGU shown in Figure 4-11(a). The conditional constraint CA of <-4,B> derived by the conditional unordered reduction. The conditional constraint specifies that the dispatcher must wait to execute C for at least 4 time units after A is executed or until B is executed. However, the contingent edge CA specifies that B will never execute before 5 time units. Therefore, by the unconditional unordered reduction, the conditional constraint CA is converted into a requirement edge CA of distance -4. Figure 4-11(b) shows the resulting CDGU after applying this requirement constraint to the distance graph.

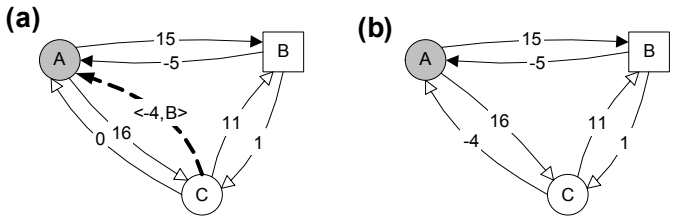


Figure 4-11 (a) A CDGU with conditional constraint CA <-4,B> where lower bound of the uncontrollable duration, 5, is greater than the wait period, 4, of the conditional constraint (b) The unconditional unordered reduction converts the conditional constraint CA of <-4,B> in to a requirement constraint CA of -4.

In this section we reviewed three reductions[4] for triangular STNUs: the precede, conditional unordered, and unconditional unordered reductions. These reductions prevent the dispatcher from squeezing the execution window of the contingent timepoint, while allowing dispatcher to react to the uncertain execution time of the contingent timepoints. If the reductions do not violate the pseudo-controllability of the STNU, then the triangular STNU is dynamically controllable [Morris 2001]. For STNUs of more than three timepoints, the triangular reductions are applied for each triangle that appears in the STNU.

In the next subsection, we introduce a technique, called *regression,* which allow us to determine if the introduction of a conditional constraint violates the pseudo-controllability of the STNU. Regression also serves to enable us to handle conditional constraints for STNUs of more than three timepoints.

4.3.2 Regression of Conditional Constraints

[Morris 2001] showed that conditional constraints need to be propagated through the distance graph. The propagation is a type of constraint propagation that resolves the conditional constraint with the other constraints in the plan. The propagation of a conditional constraint is called *regression.* This regression serves two purposes. First it detects if the conditional constraint is inconsistent with the other constraints of the plan,

[4] [Morris 2001] also introduced a general unordered reduction; however, it is unnecessary.

and second, it ensures that the conditional constraint will not be violated at execution time.

Example 4-8

Consider the distance graph shown in Figure 4-12(a). The conditional constraint CA of <-7,B> may be inconsistent if D propagates an upper bound to C that is less than 7 time units. At execution time, if D is executed at a time before 5 time units, then the propagation through DC requires C to be executed before 7 time units; hence, violating the lower bound imposed by the conditional constraint CA. However, if we impose a conditional constraint DA of <-5,B>, thereby restricting the execution time of D as shown in Figure 4-12(b), the original conditional constraint CA can not be violated. Note that the constraint DA that restricts the execution time of D only needs to be conditional because, once B is executed, the original conditional constraint CA is relaxed; thus it no longer needs to be protected. Also note that the new conditional constraint DA is computed using a similar method to that used for requirement constraints; the value of conditional constraint is equal to the shortest path DCA.

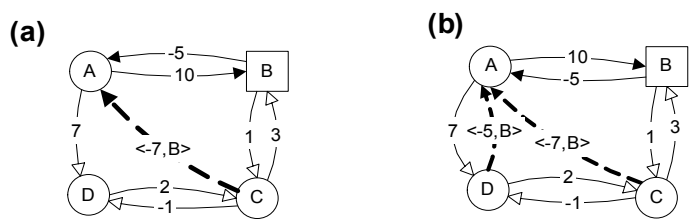


Figure 4-12 (a) The conditional constraint CA is potentially violated by the incoming positive edge DC (b) Imposing a conditional constraint of DA of <-5,B> prevents the original CA from being violated at execution time.

In general, a conditional constraint CA is potentially violated by incoming positive edges in the timepoint C. For a Conditional Distance Graph with Uncertainty (CDGU), there are two types of positive incoming edges: requirement and contingent edges. Note that conditional edges are always negative (any positive conditional edge is converted into a requirement edge by the unconditional unordered reduction). The regression lemma below specifies the means to resolve the potential consistency violations for both cases. For the requirement edge, the conditional edge is regressed using a type of shortest path computation, as illustrated in the previous example. For a contingent edge, the conditional edge is regressed using a slight variation of the precede reduction. For the contingent case, the conditional edge must be regressed, in order to ensure that it will be satisfied for all situations. The regression lemma stated below is a variation of the regression lemma introduced by [Morris 2001].

(Regression): *Given a conditional constraint CA of <B,-t>, where t is less than or equal to the upper bound of AB. Then (in a schedule resulting from a dynamic strategy):*

i.) If there isa requirement edge DC with distance w, where $w \geq 0$ and $D \neq B$, we can deduce a conditional constraint DA of <w-t, B>.

ii.) If t >= 0 and if there is a pair of contingent edges DC, of distance y, and CD, of distance -x, where x, y ≥ 0 and B ≠ C, then we can deduce a conditional constraint DA of <x-t, B>.

The first regression rule is applied when a conditional edge is threatened by an incoming positive requirement edge. The conditional edge is regressed through the incoming positive requirement edge, except when the requirement edge originates from timepoint B (i.e. D = B)[5]. The regression ensures that the wait period encoded in the conditional constraint CA of <B,-t> is never in conflict with an upper bound propagated by the incoming positive edge. The conditional constraint does not need to be regressed through an edge originating from B because, in order for the dispatcher to propagate an upper bound from B, B must be executed. When B is executed, the conditional constraint CA is relaxed (i.e. the temporal requirement is removed from the plan). The upper bound propagated from B can not be inconsistent with a constraint that is no longer exists.

If we were to regress a conditional edge CA of <B,-t> through an edge originating from B, the regression produces a new conditional constraint BA of <B,-x>. This new conditional constraint, BA, would require B to wait x amount of time after A executes or until B executes. The constraint imposes a nonsensical constraint in which B is waiting on itself to execute. One could argue that this constraint precludes B from executing until the full wait period of x as come to pass or one could argue that simply executing B satisfies the constraint; therefore, the conditional constraint is satisfied no matter when B executes. Rather than engaging in a philosophical debate, we simply restrict the regression such that this type of constraint never arises.

The second regression rule is applied when a conditional constraint CA of <B,-t> is threatened by an outcome of an uncontrollable duration. If an uncontrollable duration DC ∈ [x,y] occurs early, such that the execution of C happens before the imposed wait period of t expires, then the conditional constraint is violated. The regression imposes a new conditional constraint on the start of the uncontrollable duration, timepoint D, in order to ensure that the original conditional constraint CA will be satisfied for all situations. The conditional edge is satisfied in all situations if it is satisfied in the worst situation. The worst situation occurs when uncontrollable duration DC occurs at its earliest possible time (i.e. at its lower bound of x). Imposing a conditional constraint of DA <x-t, B> ensures that even when an uncontrollable duration occurs at its lower bound, the conditional constraint CA will not be violated. The distance (x-t) of the new conditional constraint DA is derived by treating the conditional edge as a requirement edge and applying the precede reduction.

Note the distance of the original conditional constraint CA is always less than zero (if distance is positive, then the conditional constraint is converted into a requirement edge per the unconditional reduction rule). Therefore, A must occur before C and the precede reduction rule applies. Therefore, the new conditional constraint applied through regression is only conditioned on the outcome of B. In other words, applying the

[5] [Morris 2001] did not include this exception.

unconditional reduction to positive conditional constraints prevents the regression from introducing a conditional constraint that is conditioned on more than one timepoint.

Regressions are applied recursively until no more regressions are possible. This process is called *full regression.* Each conditional edge introduced by the conditional unordered reduction needs to be regressed through all incoming positive edges. The regression of a conditional constraint through an incoming positive edge leads to either a new conditional constraint or a new requirement constraint (after applying the unconditional unordered reduction). In general, if the regression introduces a new conditional constraint, then that new conditional constraint needs to be regressed. A new conditional constraint does not need to be regressed under three cases: 1) The new conditional constraint is converted into a requirement edge by the unconditional unordered reduction, 2) the new conditional constraint is self looping (the start and end timepoint of the conditional edge are the same) or 3) there are no incoming positive edges to necessitate further regression.

One interesting case arises when the conditional constraint is converted into a positive requirement edge by the unconditional unordered reduction. If the new requirement edge is positive, then it potentially violates a conditional edge. In this case, any conditional constraint threatened by this new positive requirement edge must be regressed through it.

The regression may expose a temporal inconsistency. Specifically, if the regression imposes a self-looping (conditional or requirement) edge with negative distance (i.e. a negative cycle), then the plan constrained by the CDGU is not dynamically controllable. Note that full regression is not in itself sufficient to determine the dynamic controllability of the plan. The regression may introduce a new requirement edge that compromises the pseudo-controllability of the plan; however, it is only detected by resolving the new requirement edge with all the other constraints in the plan. Regression only resolves this new requirement constraint with the conditional constraints of the plan. The mechanism used by [Morris 2001] to detect the potential consistency violations is to recompute the APSP-graph and to recheck if the plan is pseudo-controllable. In the next section, we present a novel scheme to interleave the constraint propagation of requirement constraints with conditional constraints. This new scheme does not depend on recomputing the APSP-graph.

Example 4-9

Consider distance graph shown in **Figure 4-13**(a). In order to prevent the execution window of the contingent timepoint B from being squeezed at execution time, we apply the conditional unordered reduction to the triangle ABC. This introduces a conditional edge CA of <-7,B>, as shown in **Figure 4-13**(b). Note that other reductions are applicable, including the conditional unordered reduction on triangle DCB; however, these reductions are not applied for clarity.

This conditional edge CA needs to be regressed through all incoming positive requirement edges not originating from the conditional timepoint B, and any uncontrollable durations terminating on C. In our example, the conditional edge CA is

regressed through the requirement edge AC, and the uncontrollable duration DC. The regression through AC with distance 9 results in a new self looping conditional edge AA of <1,B>. The regression of the conditional edge CA through the uncontrollable duration DC results in a new conditional edge DA of <-4,B>. The results of these regressions are shown in **Figure 4-13**(c).

This conditional edge AA is converted into a requirement edge by the unconditional unordered reduction, because the distance of the conditional edge is positive. Fortunately, this new requirement edge does not introduce a negative cycle into the CDGU. The distance of the conditional edge DA is -4, which imposes a wait of 4 time units between A and D, which is less than the lower bound of the uncontrollable duration AB of 5. Therefore, the conditional constraint DA of <-4,B> it is converted into a requirement edge DA with distance 4 by the unconditional unordered reduction. The results of these reductions are shown in **Figure 4-13**(d). Note that there now exists a negative cycle between AD; however, this is not detected during regression.

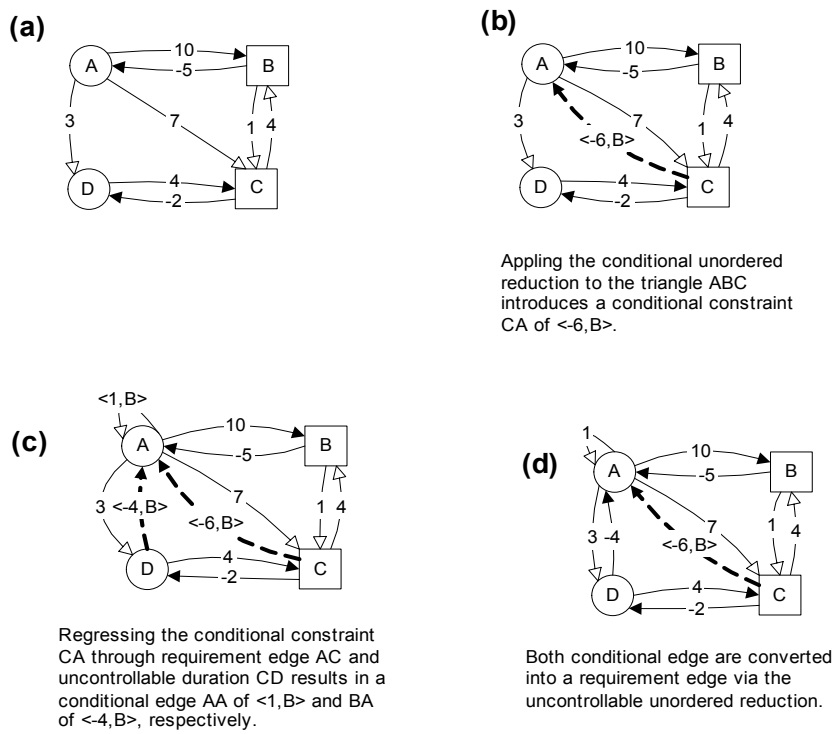


Figure 4-13 (a) A four timepoint DGU. (b) The CDGU after appling the conditional unordered reduction to triangle ABC. (c) The CDGU after regressing the conditional edge CA through AC and DC. (d) The CDGU after converting the conditional constraints to requirement edge via the unconditional unordered reduction

Several important patterns arise during regression. First, all conditional edges introduced by regressing a conditional edge CA of <B,-t> always points to C and the regression rules prevent a conditional edge of BA; therefore, there is at most N-1 conditional edges conditioned on B. For a plan containing P uncontrollable durations,

there can be at most P*(N-1) conditional constraints in the plan. Second, the only way two conditional constraints can exist between the same timepoints is when the uncontrollable durations start the same timepoint. Third, the regression always increases the temporal distance of the conditional constraint (i.e. progressively imposes a less restrictive constraint).

In this subsection we presented a constraint propagation technique, called regression, that enabled us to resolve the conditional constraint with other constraints in the plan. Regression enabled the ternary conditional constraints to be propagated similar to simple requirement edges. In the next subsection, we combine a pseudo-controllability checking algorithm, with the triangular reductions and regression, to form the dynamic controllability algorithm.

4.3.3 Pseudo-Code for the Dynamic Controllability Algorithm

The following completes the description of the dynamic controllability (DC) algorithm [Morris 2001] by presenting the pseudo-code. The pseudo-code for the DC algorithm is shown in **Figure 4-14**. Dynamic controllability transforms the STNU into a dispatchable CDGU, if this reformulation is successful, then the algorithm is dynamically controllable and returns true, otherwise the DC algorithm returns false. Recall that the general structure of the DC algorithm is described in the flow diagram shown in **Figure 4-5**.

Line 1 computes the associated distance graph, G, of the STNU, Γ. The DC uses the distance graph formulation of temporal constraints. In Line 3 the DC algorithm first computes the All-Pair Shortest-Path graph (APSP-graph) of the distance graph G while ignoring the distinction between contingent and requirement edges. Line 4 checks if the plan is pseudo-controllable by calling the IS_PSEUDO_CONTROLLABLE function. If any contingent edges are squeezed or if any negative distance graph contains a negative cycle, then the IS_PSEUDO_CONTROLLABLE function returns false; otherwise, it returns true. If the plan is not pseudo-controllable, the DC algorithm returns false. Recall that if the plan is not pseudo-controllable, then the plan is not dynamically controllable. Line 6 initializes the variable modified to false. This variable is used to determine if the algorithm converges.

Lines 7-14 loops through all possible triangles, ABC, that contain a contingent timepoint B, and applies any tightening required by the precede reduction, and any conditional constraint required by the conditional unordered reduction. If the reductions tighten or add a new constraint to the distance graph, then the algorithm assigns the variable modified to true and breaks out of the loop. If the algorithm loops through all possible triangles and the constraints of the distance graph are not modified, the variable modified remains false.

In Line 15 The REGRESS_WAITS function applies all possible regressions of conditional constraints, while converting the conditional constraints to requirement constraints when the unconditional unordered reduction applies. If the regression introduces a temporal inconsistency, then the REGRESS_WAITS function returns false;

otherwise, it returns true. If the REGRESS_WAITS function returns false, then so does the DC algorithm in Line 16.

Line 17 checks if the regression modified the constraints (conditional, requirement, or contingent) of the distance graph by calling the function IS_MODIFIED. The variable modified is true if the distance graph is modified, by either applications of the reductions or regression.

The algorithm loops through Lines 2-17, tightening the edges of the distance graph until the algorithm converges on a dispatchable CDGU (the algorithm completes on a successful loop when no edges are modified) or loops until the algorithm detects that the plan is not dynamically controllable (either in Line 5 or Line 16).

```
function DC1(Γ)
input A STNU Γ
effects computes a dynamically controllable CDGU if the plan is dynamically controllable.
returns true if Γ dynamically controllable, otherwise false
1  G ← DISTANCE_GRAPH(Γ)
2  do
3     G ← COMPUTE_APSP_GRAPH(G)
4     if ¬IS_PSEUDO_CONTROLLABLE (G)
5        return FALSE
6     modified ← FALSE
7     for each contingent timepoint B ∈ N(G) associated with uncontrollable duration AB
8        for each incoming positive edge CB
9           modified ← apply tightenings required by the precede reduction to triangle ABC.
10          modified ← apply conditional constraints required by conditional unordered reduction
                       to triangle ABC
11          if modified break
12       end for
13       if modified break
14    end for
15    if ¬REGRESS_WAITS(G)
16       return FALSE
17    modified ← IS_MODIFIED(G) or modified
16    end if
17 while modified=TRUE
18 return TRUE
```

Figure 4-14 Pseudo-Code for Dynamic Controllability (DC) algorithm [Morris 2001]

The DC algorithm is sound because it only derives new constraints based on the original constraints and the assumption of dynamic controllability [Morris 2001]. Furthermore, the completeness of this algorithm was shown in [Morris 2001].

The time complexity of the DC algorithm is shown to be polynomial [Morris 2001]. The individual tightenings are clearly polynomial, and convergence is assured because the domains of the constraints are strictly reduced by the tightenings. However, only a crude estimate was provided for how long the convergence would take. Moreover, a crude estimate is in terms of the maximum value of the edges and the fixed precision on the edges. Each time the algorithm loops through Lines 2-17 it applies at least one tightening. If all the distance on the edges are bounded by $\pm$ B, and there is a fixed level of precision δ, and E edges. Then, after at most BE/δ loops, the algorithm will converge. Each loop requires an $O(N^3)$ APSP computation and there are N^3 edges in the APSP graph; Therefore, the crude bound becomes N^6*B/δ !

The DC algorithm depends on repeated calls to an expensive $O(N^3)$ APSP computation in Line 2 to perform requirement edge constraint propagation. Furthermore, it uses an inefficient looping scheme that first resolves the requirement edges with one another via the APSP algorithm, then resolves the requirement edge with the contingent edge propagation via reductions, and, finally, resolves the conditional edges with the requirement, and conditional edges with contingent edges via regression. In the next section, we show how to improve on the performance of the dynamic controllability algorithm by resolving all possible combinations of constraints all at once. This general frame work enables our new fast dynamic controllability algorithm to remove the repeated APSP computations.

4.4 Fast Dynamic Controllability Algorithm

In this section, we describe our novel fast dynamic controllability algorithm (fast-DC algorithm). This fast-DC algorithm has a significant performance improvement compared to the dynamic controllability algorithm introduced by [Morris 2001]. This fast dynamic controllability algorithm achieves its enhanced speed via several new improvements. The speed of the fast-DC algorithm is verified empirically.

1. We show how to exploit the fact that a dispatchable plan can by incrementally executed during the reformulation phase. We introduce a local incremental algorithm for maintaining the dispatchability of a plan constrained by STNs. In this algorithm, when an edge length changes in a dispatchable distance graph, only a subset of the constraints need to be notified of this change. Specifically, the change only needs to be *back-propagated*, similar to regression. Then we show how to apply this technique to plans constrained by STNUs. In order to make this transition from STNs to STNUs, we introduce a new property, called *pseudo-dispatchability*, and show that for any for pseudo-dispatchable STNU only changes in requirement edges need to be back-propagated. This removes the need to compute the APSP-graph, which updates all edges in the distance graph, every time a requirement edge changes.

2. The constraint propagations of requirement edges, contingent edges and conditional edges required by dynamic controllability are combined into an efficient general framework. This general framework enables the different types of constraint propagation to be interleaved with one another rather than applying them sequentially. Interleaving the different types of propagation enables the dynamic controllability algorithm to reduce the number of propagations required. The idea is to apply the required tightening as soon as we can deduce them, so that the next round of propagations has the most up-to-date constraint set as possible.

3. We trim the distance graph of redundant constraints prior to performing the integrated constraint propagation. This can drastically reduce the number of propagations required.

First we introduce the incremental algorithm for maintaining the dispatchability of STNs. Next we show how this incremental algorithm applies to STNUs by introducing the idea of pseudo-dispatchability. This provides the basis for the new requirement edge propagation technique, which removes the dependence of the dynamic controllability algorithm on repeated APSP calls. Next, we describe the set of back-propagation rules that make up the general constraint propagation framework and present the back-propagation algorithm. Finally, we describe the new fast-DC algorithm pseudo-code, which uses this new back-propagation algorithm. After presenting the algorithm, we demonstrate the fast-DC algorithm on several examples and review how the fast-DC algorithm fits in with the Hierarchical Reformulation algorithm, presented in Chapter 3.

4.4.1 Incremental Dispatchability Maintenance

In order to understand how the new requirement constraint propagation technique works, let's revisit the problem of dynamically executing a STN. [Muscettola 1998a] showed that any dispatchable STN can be executed incrementally using a dispatching algorithm. If a STN is dispatchable, as long as each execution decision is consistent with the past assignments, then we can guarantee that there is a consistent assignment for future timepoint assignments. Recall that executing a timepoint is equivalent to adding a set of rigid constraints between the start of the plan and the timepoint being executed. During execution, the dispatcher ensures that the addition of these additional constraints is consistent with the past, by propagating information at execution time. However, if a random constraint is modified in a dispatchable graph, we need to make sure that the change is consistent with the past using *back-propagation*. Back-propagation informs all constraints that relate timepoints in the past. If the back-propagation does not introduce an inconsistency, then the constraint change is consistent with all the constraints. This leads to an efficient algorithm for incrementally updating a dispatchable STN distance graph.

In order to develop a back-propagation algorithm for a dispatchable STN, we use a logic similar to that used when developing the reduction and regression rules. Specifically, any positive edge AB that is either added or modified is only threatened by

outgoing negative edges from B. In addition, any negative edge BA that is either added or modified is only threatened by incoming positive edge to A. Therefore, we need to back-propagate any change in a positive edge AB through all negative edges originating from B. Similarly, we need to back-propagate any change in a negative edge BA through all incoming positive edge into B.

These back-propagations need to be applied recursively in order to ensure that the change is consistent with the past. This back-propagation technique only requires us to update a subset of the edges (i.e. constraint that may happen in the past), instead of all the edges which would happen if we were to recompute the APSP every time an edge changed. The future constraint will be notified of the change when they need to be notified, which is at execution time. Thus we defer the future updates until execution time. Furthermore, the back-propagation preserves the dispatchability of the distance graph.

First, we give an example, then we provide the formal back-propagation rules for distance graphs associated with STNs. Finally, we show how this back-propagation is extended to distance graphs associated with STNUs.

Example 4-10

Consider the dispatchable distance graph shown in Figure 4-15. Figure 4-15(a) shows the original dispatchable graph. Consider a scenario in which the edge DC is reduced from -2 to -5 for some reason, as shown in Figure 4-15(b). During execution, the edge DC is used to propagate a lower bound to timepoint D. We call timepoint D the timepoint of interest. In order to maintain the dispatchability of the graph, the tightening of DC needs to be propagated through the graph. However, the effects only need to be propagated backward from the node of interest, because, as long as D is consistently executed, the dispatcher is able to consistently execute E.

The negative edge DC is threatened by the incoming positive edge CD and BD. We resolve the new edge DC with the threats (CD and BD), by computing the local shortest path through the threats. The shortest path BDC results in a tightening of edge BD from 8 to 5, and the shortest path CDC results in a new edge CC of 5. Figure 4-15(d) shows the result of the first step of back-propagation. The tightening of the constraint BC is then back-propagated where node C is the timepoint of interest. The edge positive BC is threatened by all outgoing negative edges from C (CB and CA), as shown in Figure 4-15(e). BC is back-propagated through its threats. The shortest path BCB results in a new edge BB of 0, and the shortest path BCA results in a tightening of BA from 0 to -1. The results of the second stage of back-propagation are shown in Figure 4-15(f). Note that BA needs to be back-propagated through AB, resulting in a new edge AA of 9.

Back-propagation does not introduce any negative cycles; therefore, the change is consistent. Furthermore, the tightenings introduced through the back-propagation preserve the dispatchability of the distance graph.

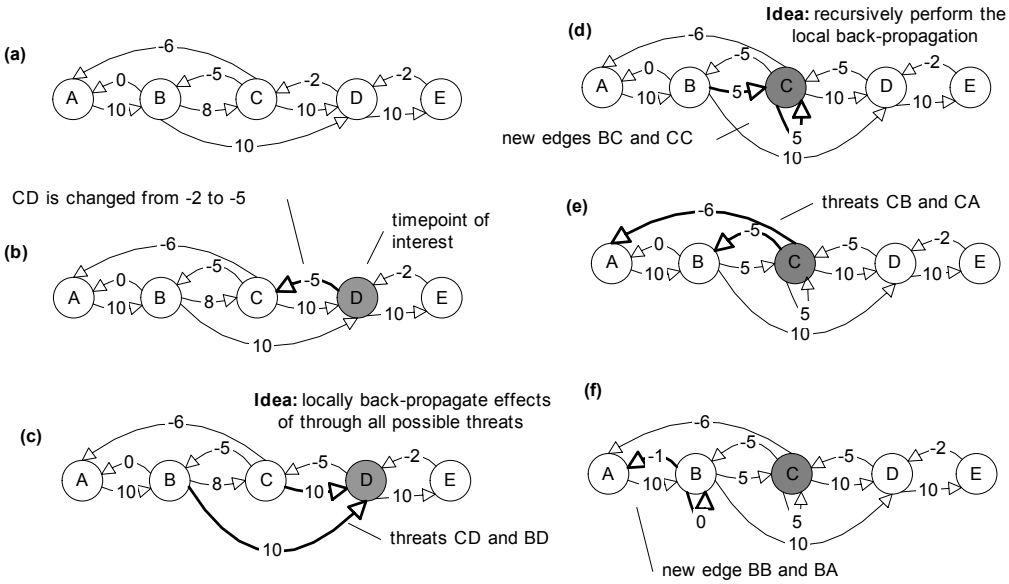


Figure 4-15 Back-Propagation Example

Now we give the back-propagation lemma and incremental algorithm used to maintain the dispatchability of a STN distance graph.

Lemma (Incremental Dispatchability) *Given a STN and associated dispatchable distance graph G,*
i) any change or addition of an edge AB of distance x, where x >0, for all edges BC of length y, where y <= 0, we can deduce a new constraint AC of length x+y.
ii) any change or addition of an edge BA of distance z, where p <= 0, for all edge CB of length q, where q >= 0, we can deduce a new constraint CA of length p+q.
Furthermore, recursively applying rules i and ii maintains the dispatchability of the G.

The algorithm for maintaining the dispatchability of the distance graph, recursively applies the Incremental Dispatchability propagation rules until no more back-propagations can be deduced. If back-propagation introduces a negative cycle then the algorithm returns false, otherwise, the algorithm returns true

The Incremental Dispatchability (ID) algorithm used to replace the APSP computation in the slow dynamic controllability algorithm introduced by [Morris 2001]. In order to apply the ID algorithm to distance graphs with uncertainty (DGUs) we introduce the idea of *pseudo-dispatchability.* If we ignore the distinction between contingent and requirement edges in the DGU (as we did when we computed pseudo-controllability), then the DGU is effectively converted into STN distance graph. If this associated STN distance graph is dispatchable, then we say the DGU *pseudo-dispatchable.* In order to maintain the pseudo-controllability of DGU when a constraint is changed, we apply the ID algorithm to the DGU. This resolves a change in a requirement constraint with all the other requirement constraints.

We also introduce the term *pseudo-minimal dispatchable graph (PMDG).* A PMDG is DGU in which the associated STN distance graph contains the fewest number of edges. The edges of the DGU are trimmed using the same dominance relationships introduced by [Muscettola 1998a].

4.4.2 Back-Propagation

In this subsection we describe a set of local constraint propagation rules that determine how one constraint change affects the values of other constraints, in order to maintain the dispatchability of a dynamic controllability Conditional Distance Graph with Uncertainty (CDGU). These rules all share one important property - they only affect constraints that occur earlier in the plan; thus, we call them *back-propagation* rules for STNUs. This idea is illustrated in **Figure 4-16**. These rules and the associated back-propagation algorithm form the basis of the Fast-DC algorithm. The back-propagation rules integrate the Incremental Dispatchability rules, the reduction rules and the regression rules. The back-propagation rules put all these rules in to common framework.

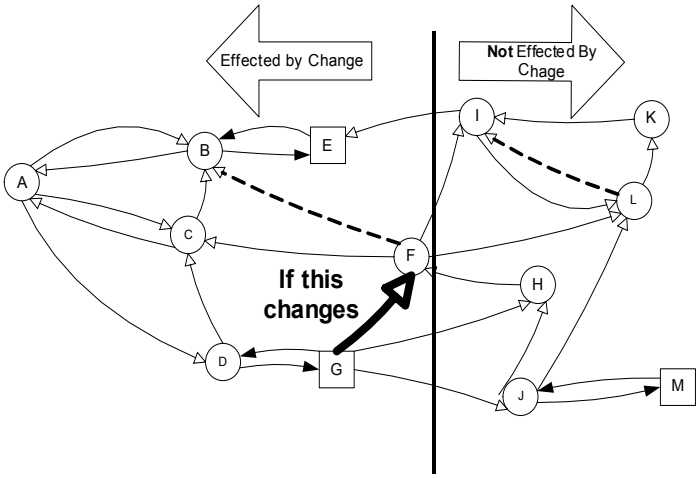


Figure 4-16 If either a requirement, or conditional edge changes, in order to maintain the dispatchability of the CDGU, the effects only need to be back-propagated

Each back-propagation rule differs, depending on the types of edges involved, the signs of the edge distances, and the relative direction of the edges. In a DGU, there exist five types of edges: positive and negative requirement edges, positive and negative contingent edges, and negative conditional edges.

Requirement	Contingent	Conditional
+ -	+ -	-

We group our back-propagation rules into three groups: negative requirement edges, positive requirement edges, and negative conditional edges because these are the only three types of edges that may be added or modified during reformulation. Any positive conditional edge is converted to a requirement edge by the unconditional unordered reduction rule. The rules are used to determine what new constraints need to be enforced to ensure consistency and dynamic controllability. The following table summarizes the back-propagation rules used in the Fast-DC algorithm.

If This Changes	**Must Back-Propagated Through**	**Derived From**
Negative requirement edge BA	1. any positive requirement edge CB	ID(i)
	2. any positive contingent edge CB	Precede Reduction
Positive requirement edge AB	1. any negative requirement edge BC	ID(ii)
	2. * any negative contingent edge BC	CUR
	3. any negative conditional edge BC of <-t,D> where D ≠A	Regression(i)
Negative conditional edge BA	1. any positive requirement edge CB of <-t,D> where D ≠A	Regression(i)
	2. any positive contingent edge CB	Regression(ii)

* apply conditional constraint in both precede or unordered cases
ID: Incremental Dispatchability
CUR: conditional unordered reduction

Table 1 Back-Propagation Rules Summary

4.4.3 Back-Propagating when a Negative Requirement Edge Changes

Recall that when a dispatcher executes a timepoint it propagates that execution times through the distance graph in order to update the execution windows of the neighboring nodes. The dispatcher uses the negative edges to update the lower bound of the timepoint's execution window. The only way a timepoint's lower bound, derived from a negative edge propagation, can be violated is if some other positive edge propagates an upper bound that is smaller than this lower bound. The back propagation rules are used to prevent this inconsistent condition from happening.

Recall that there are two types of positive edges in the DGU: a positive requirement edge (Case1) and a positive contingent edge (Case2). The back-propagation rule for changing negative requirement edges handles both cases. In [Morris 2001], the first case was handled by the APSP computation, and the second case was handled by the precede reduction.

Case 1: Back-propagating a negative requirement edge through a positive requirement edge.

This back propagation rule is called in case when there exists some change in or creation of a negative requirement edge BA with weight *a*, such that there exist some positive requirement edges CB with weight *b*. The back-propagation rule derives a new constraint with CA with weight a+b. If this new edge CA provides a tighter constraint then it update the DGU accordingly. Note that the arbitrary timepoint C may be the timepoint B, in which case the derived constraint loops on the timepoint B. This example is depicted in **Figure 4-17** (Case1).

Case 2: Back-propagating a negative requirement through a contingent link.
This is exactly the same case as the precede case derived in the dynamic controllability algorithm [Morris 2001]. This propagation is illustrated in **Figure 4-17** (Case2). The correctness of this propagation rule is shown in [Morris 2001], for the case where there exists some negative requirement edge.

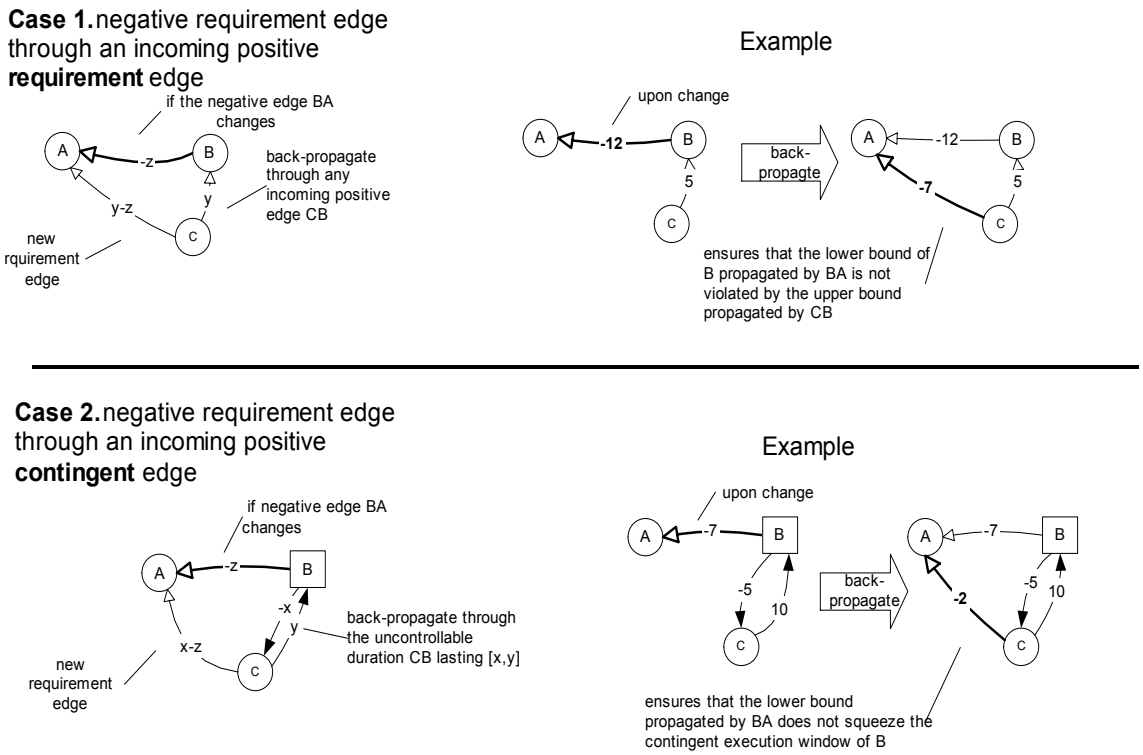


Figure 4-17 Back-Propagation Rules for Negative Requirement Edge

4.4.4 Back-Propagation Rule when Positive Requirement Edge Changes

If a positive requirement edge changes, there are three cases to consider. All three cases are illustrated in **Figure 4-18**. The rationale for each rule is shown in Table 1.

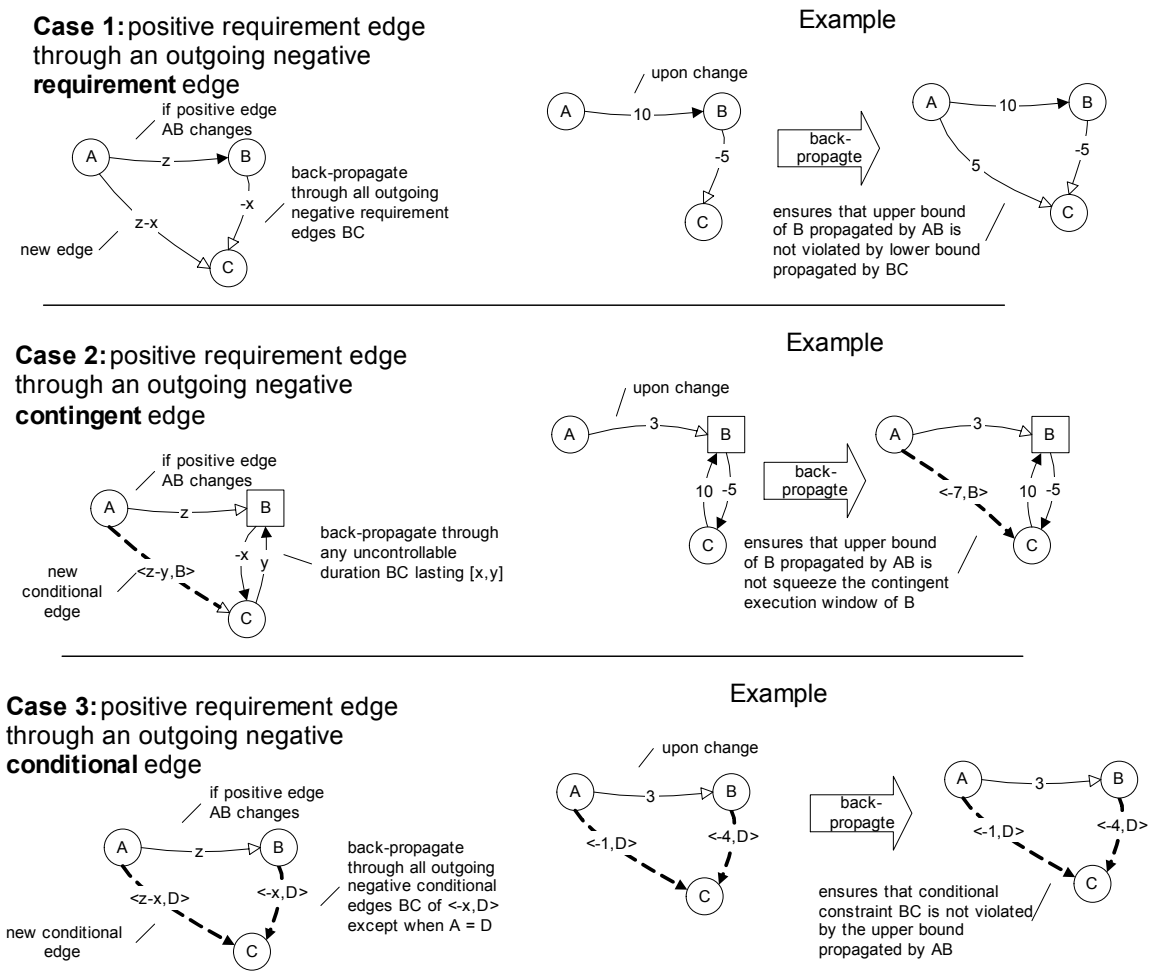


Figure 4-18 Back-Propagation Rule for Positive Requirement Edges

4.4.5 Back Propagating Conditional Edges

The back-propagation rule for the addition or change of a conditional constraint is exactly the same as the regression rules. The back propagation rules are shown here for completeness. They are illustrated in Figure 4-19.

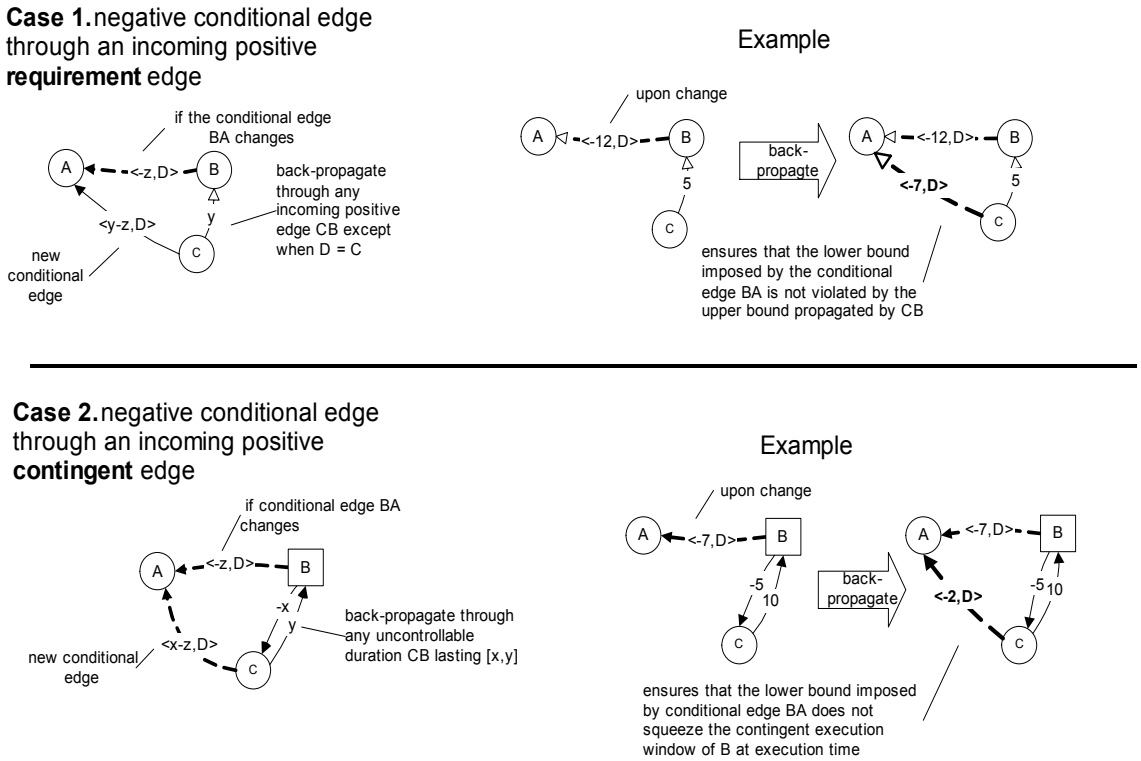


Figure 4-19 Back-Propagation Rules for Conditional Edges

5.1.1 Pseudo-Code for BACK-PROPAGATE

The pseudo-code for BACK-PROPAGTE is show in Figure 4-20. BACK-PROPAGATE is a function that recursively applies the back propagation rules in the previous sections. It accepts a CDGU G, a start timepoint u, and an end timepoint v. BACK-PROPAGATE is initiated in order to prevent an positive requirement edge (u,v) from squeezing the upper bound of the contingent timepoint *v* or in order to prevent a negative requirement edge (u,v) from squeezing the lower bound of a contingent timepoint *u*. The algorithm returns true if the edge (u,v) is successfully back-propagated through G. (i.e. no inconsistencies were introduced) otherwise the algorithm returns false.

Lines 1-2 detect two possible termination conditions. If the timepoint u = v, the edge(u,v) is a loop. If this loop is positive, thus, does not introduce a temporal inconsistency, then the algorithm returns true in Line 1. However, if the loop is negative then the algorithm returns false in Line 2

Lines 3-15 applies the all applicable back-propagations associated with edge (u,v). Specifically the algorithm back-propagates (u,v) through all appropriate edge (x,y) resulting in a new edge (p,q). In line 5 it applies the unconditional unordered reduction when appropriate, which converts a conditional edge (p,q) into a requirement edge (p,q). This new edge (p,q) (conditional or requirement) is resolved with G. If G is modified it does two things. It checks if the new edge (p,q) introduces any local negative cycles. Specifically, it checks if the cycle p-q-p is negative. If there is a local negative cycle, then the algorithm returns false, otherwise the algorithm recursively calls BACK-PROPAGATE(G, p, q). If this BACK-PROPAGATE returns false, then the orginal BACK-PROPAGATE function returns false. If the algorithm successfully applies all possible back-propagations of (u,v) in line 3-13, then the algorithm returns true in Line 16.

In the next section we give a example of the BACK_PROPAGATE function in the context of the Fast Dynamic Controllability algorithm.

```
Method BACK-PROPAGATE(G,u,v)
Input: CDGU G, start timepoint u, and end timepoint v
Effects: recursively called function that back-propagates the constraints through G
Returns: true if the no inconsistencies where introduces, otherwise false
    if IS-POS-LOOP( u, v )  return TRUE
    if IS-NEG-LOOP( u, v ) return FALSE
    for each edge (x,y) where the back-propagation rules apply to edge (u,v)
        back-propagate (u,v) through (x,y) to create new edge (p,q)
        convert any conditional constraint (p,q) to a requirement edge (p,q) as required
        by the unconditional unordered reduction
        resolve the edge (p,q) with G  by tightening (or adding) corresponding edge (p,q) in G
        if G is modified
            if resolving (p,q) with G  introduces a local negative cycle
                return FALSE
            end if
            if ¬BACK-PROPAGATE(G,p,q)  // recursive call
                return FALSE
            end if
        end if
    end for
    return TRUE
```

Figure 4-20 Pseudo-Code for BACK-PROPAGATE

4.5 Fast Dynamic Controllability Pseudo-Code

The pseudo-code for the Fast-DC algorithm is shown **Figure 4-22**. The algorithm is used to reformulate the group plans in the Hierarchical Reformulation Algorithm. The Fast-DC algorithm operates on group plan's associated STNU. If the STNU associated with the group plan is dynamically controllable, then the algorithm returns a pseudo-minimal dispatchable CDGU, which can be directly executed by the STNU dispatching algorithm introduced by [Morris 2001], otherwise, the algorithm returns NIL. The description of the Fast-DC pseudo-code is interleaved with a example. The TPNU used in the example is shown in Figure 4-21. This group plan is part of the Mars rover example originally introduced in Section 3.4.

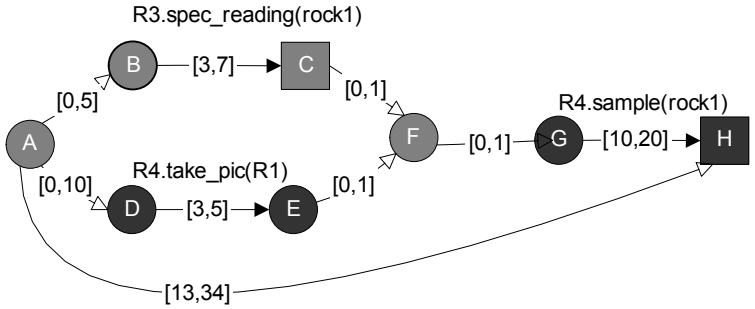


Figure 4-21 Sample Group Plan

---

function FAST-DC(Γ)
input: A Simple Temporal Network with Uncertainty Γ
returns minimal dispatchable CDGU if Γ is dynamically controllable, otherwise NIL
1 $G \leftarrow$ STNU_TO_CDGU(Γ)
2 **if** ¬COMPUTE_MPDG(G)
3 **return** NIL
4 **end if**
5 **if** ¬ IS_PSEUDO_CONTROLLABLE (G)
6 **return** NIL
7 **end if**
8 S ← start timepoint of G
9 Bellman_Ford_SDSP(S, G)
10 Q ← ordered list of contingent timepoints according to the SSSP distances
11 **while**(¬Q.IS-EMPTY())
12 n ← Q.POP_FRONT()
13 **if** ¬ BACK_PROPAGATE_INIT(G, n)
14 **return** NIL
15 **end if**
16 **end while**
17 COMPUTE_MPDG(G)
18 **return** G

Figure 4-22 Pseudo-Code of Fast Dynamic Controllability Algorithm (Fast-DC)

Line 1 converts the STNU into a CDGU. This conversion is trivial. It converts the links of the STNU into a pair of directed edges. Note that initially, the CDGU does not contain any conditional constraints; therefore, the original CDGU is similar to a Distance

Graph with Uncertainty (DGU). For example, **Figure 4-23** shows the CDGU of the sample group plan.

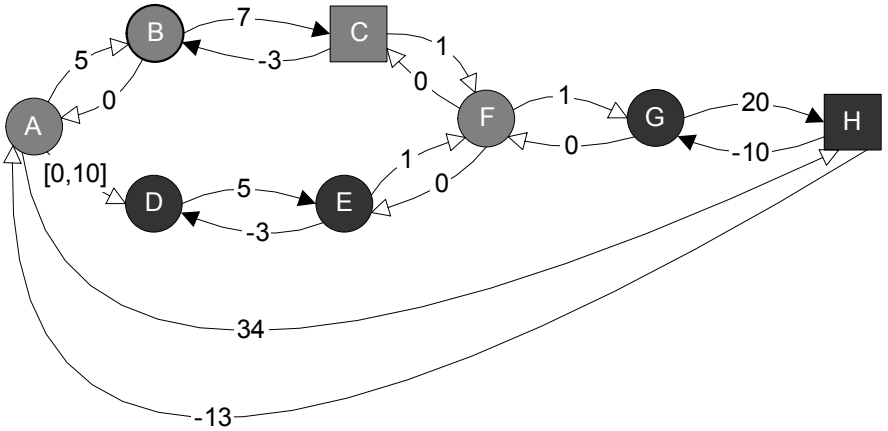


Figure 4-23 CDGU of the Sample Group Plan

Recall that in order to apply the back-propagation rules, the CDGU must be pseudo-dispatchable. In addition, in order to efficiently apply the back-propagation rules, the CDGU should contain the fewest number of edges. Line 2 transforms the CDGU into a Minimal Pseudo-Dispatchable Graph (MPDG) by calling the COMPUTE_MPDG function. This function applies the basic STN Reformulation Algorithm [Muscettola 1998a] on the CDGU. The STN Reformulation algorithm is applied by ignoring the distinction between contingent and requirement edges in the CDGU. This function reformulates the constraints of the CDGU. If the CDGU is pseudo-dispatchable, then the function COMPUTE_MPDG returns true, otherwise it return false. If the CDGU is not pseudo-controllable, then the FAST-DC algorithm returns NIL. The minimal pseudo-dispatchable graph for the sample group plan is shown in **Figure 4-23**.

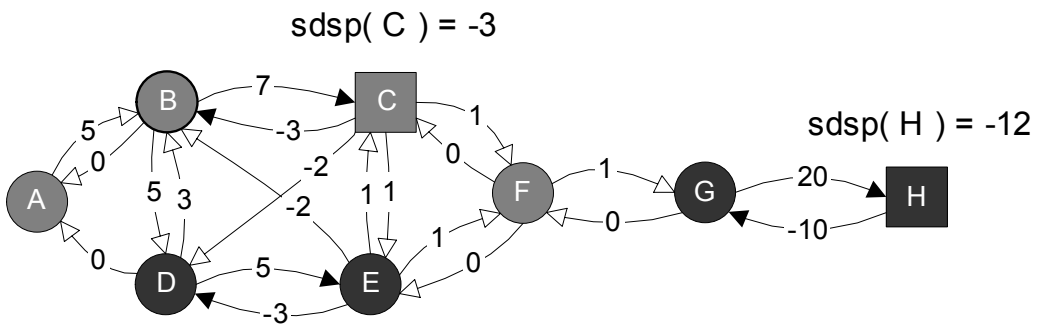


Figure 4-24 MPDG of the Sample group plan

The CDGU is only dynamically controllable if it pseudo-controllable. Recall that if a graph is pseudo-controllable then the constraints do not strictly imply a tighter constraints on the contingent edges. Lines 5-7 of the FAST-DC algorithm checks if the contingent edges are squeezed during the process of converting the CDGU into a minimal pseudo-dispatchable graph. If the CDGU is not pseudo-controllable, then the FAST-DC algorithm returns NIL. In our example, contingent edges BC, CB, GH, and HG all remain unchanged; therefore, the CDGU is pseudo-controllable.

Recall that our goal is to reformulate the graph to ensure that the plan can be dynamically executed. This reformulation is done by applying the a recursive BACK_PROPAGATE function. The BACK-PROPAGATE function needs to be applied to any edge that may squeeze the contingent timepoint at execution time. Each initial call of BACK_PROPAGATE causes a series of other edge updates. However, they will only update edges closer to the start of the plan. In order to reduce the amount of redundant work, we initiate the back-propagation cycle near the end of the plan to the start of the plan. This way we slowly build up a solution from the end of the plan to the start of the plan. In order to organize the back-propagations, we need to create a list of contingent timepoints ordered from timepoint that are executed near the end of the plan to the timepoints that are executed near the beginning of the plan.

Lines 8-10 create this ordered list, Q, of the contingent timepoints. The contingent timepoints are ordered based on their Single-Destination Shortest-Path (SDSP) distance, sdsp(x). Specifically, the contingent timepoints are ordered from lowest to highest SDSP distances. Note that all SDSP distances are less than or equal to zero. The SDSP distances are computed in Line 9, and the contingent timepoints are ordered in Q in Line 10. The sample group plan contains two contingent timepoints C and H with sdsp(C) = -3 and sdsp(H) = -12. Therefore, timepoint H comes before timepoint C.

Lines 11-16 of the FAST-DC algorithm apply the back-propagation rules. Line 12 pops the first contingent timepoint, n, off of the list Q and calls the BACK-PROPAGATE_INIT function, which starts one round of back-propagations. If that back-

propagation round results in a inconsistency, then the FAST-DC algorithm returns NIL. The pseudo-code for the BACK-PROPAGATE_INIT is shown in Figure 4-25. BACK-PROPAGATE_INIT ensure the that contingent timepoint n is never squeezed during execution.

```
BACK-PROPAGATE-INIT( G, v )
Input: A CDGU G and contingent timepoint v
Returns: true if no inconsistencies were introduced during the back-
propagation cycle, otherwise false
1  for all incoming positive edges (u,v ) into the contingent timepoint v
2      if ¬BACK_PROPAGATE(G,u,v)
3          return FALSE
4      end if
5  end for
6  for all outgoing negative edges (v,u) from the contingent timepoint v
7      if ¬BACK_PROPAGATE(G,v,u)
8          return  FALSE
9      end if
10 end for
11 return TRUE
```

Figure 4-25 Pseudo-Code for BACK-PROPAGATE-INIT

The BACK-PROPAGATE-INIT function initiates the back-propagation by applying the back-propagation rules to ensure that the contingent timepoint *v* is never squeezed during execution. Recall the contingent timepoint can only be squeezed by incoming positive edges or outgoing negative edges. Lines 1-5 calls BACK_PROPAGATE for all incoming positive edges into the contingent timepoint *v* and Lines 6-10 calls BACK_PROPAGATE for all outgoing negative edges from *v*.

For example, consider the series of back-propagations shown in Figure XXX.
There does not contain any possible edges to violate contingent timepoint H so no back-propagation is required. For timepoint C the edge EC is back-propagated through BC resulting in a new conditional edge EB of <C,-6>. This edge then back-propagated DE which modifies the requirement edge DE to -1. This requirement edge is then back-propagated through edge BD resulting in the edge BB of distance 4. This thread of back-propagation terminates here because of a positive self-loop.

The contingent timepoint C is also threatened by the outgoing negative edge CD of length -2. This edge CD is back-propagated through BC which modifies BD = 1. This is then back-propagated through DB resulting modifying the self looping edge BB to 0. No more

propagations are necessary. The resulting dispatchable CDGU is shown in Figure 4-26. The back-propagation did not introduce an inconsistency; therefore, the sample group plan is dynamically controllable.

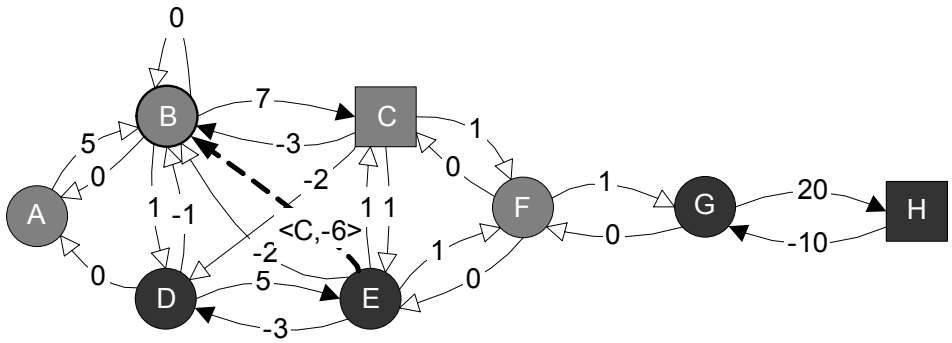


Figure 4-26 Dispatchable CDGU after back-propagation

The last step of the Fast-DC algorithm is to trim the dominated (redundant) edges from the CDGU. This is done by calling the basic STN reformulation algorithm. The resulting graph is a minimal dispatchable CDGU which can be executed by the dispatching algorithm introduced by [Morris 2001]. For example, the minimal dispatchable CDGU for the sample group plan is shown Figure 4-27.

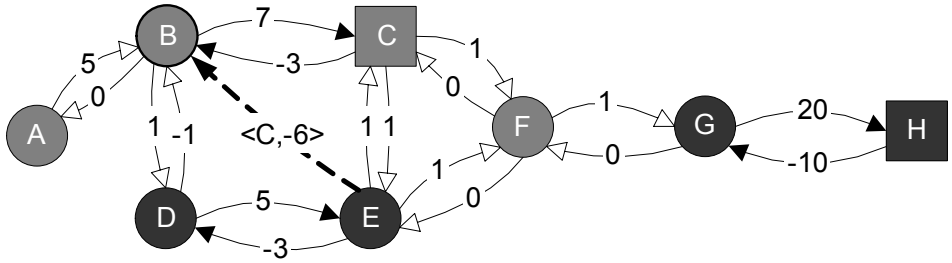


Figure 4-27 CDGU of sample group plan after trimming the redundant edge

4.6 Summary

In this chapter we reviewed the dynamic controllability algorithm introduce by [Morris 2001]. Then we generalized the reduction rules introduced to [Morris 2001] in order to develop an efficient dynamic controllability algorithm. This new Fast-DC algorithm is used by the HR algorithm presented in Chapter 3 to reformulate the group plans. In the

next chapter we present empirical data demonstrating the speed of this new Fast-DC algorithm.

5 Results and Conclusion

5.1 Introduction

The outline for this chapter is as follows. First we discuss the implementation of the Hierarchical Reformulation (HR) algorithm. Then we discuss the experimental results of the fast dynamic controllability (Fast-DC) algorithm. We then discuss the limitations of our approach and directions for future work. We conclude with a summary of the major contributions of this thesis.

5.2 Implementation of the Hierarchical Reformulation algorithm

The Hierarchical Reformulation (HR) algorithm has been implemented in C++ and tested with a variety of hand coded examples, including the cooperative Mars rover scenario presented in Chapter 3.4. In order to generate the two-layer multi-agent plans, we developed a MTPNU Graphical User Interface (GUI) implemented in Java.[6] The GUI enables the user to create and visualize the MTPNUs. The screenshot of the editor, shown in **Figure 5-1**, shows the mission plan for the Mars rover scenario described in Chapter 3.4. The editor allows the user to create, modify, and visualize the multi-agent plans in a variety of different forms. All of the plans generated for the HR algorithm were created using the MTPNU GUI.

[6] The MTPNU GUI was developed by Andreas Wehowsky and myself.