Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2011-002 January 15, 2011

Flexible Execution of Plans with Choice and Uncertainty

Patrick R Conrad and Brian C Williams

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Flexible Execution of Plans with Choice and Uncertainty

Patrick R. Conrad prconrad@mit.edu
Brian C. Williams williams@mit.edu
Room 32-227
32 Vassar St
Cambridge, MA 02139 USA

December 22, 2010

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.

1 Introduction

Model-based executives strive to elevate the level of programming for autonomous systems to intuitive, goal-directed commands, while providing guarantees of correctness. 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. Likewise, temporal plan executives take input specifications of the timing requirements of a plan and then compute a strategy for executing the plan according to the requirements. In both cases, engineers can develop a general executive and tailor it to individual systems by specifying goals appropriately, rather than needing to program a specific set of routines for each robot.

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 with the benefit of knowing what happened during earlier portions of that plan. In contrast, a system that makes all decisions before execution 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.

Muscettola, Morris, and Tsamardinos developed a technique for dynamically executing temporal plans whose events are related through simple temporal constraints [13]. At the core is a dynamic scheduling algorithm for Simple Temporal Problems (STPs). Morris, Muscettola and Vidal extended their execution model to handle Simple Temporal Problems with Uncertainty (STPUs), providing guarantees of robustness to modeled, finite bounded uncertainty [11]. In order to handle richer families of choices, later authors introduced different types of choices that the executive is allowed to make, and developed new executives to handle these models. For example, Tsamardinos, Pollack, and Ganchenv developed an executive for Disjunctive Temporal Problems, a relative of STPs that includes disjunctive constraints [24]. Kim, Williams, and Abramson introduced Kirk, an executive that dynamically select sub-plans while resolving resource conflicts that might arise during sub-plans [9]. Shah and Williams developed Chaski, an executive that can dynamically allocate tasks between agents [16].

One key feature of these executives is that they can delay making choices until run-time. Delaying decisions is useful because making decisions later means that more information is available, allowing the executive to react to real-world outcomes and make decisions with less uncertainty. This reduces the conservatism required to guarantee correctness. Furthermore, when uncertainty is explicitly modeled, dynamic executives can correctly execute plans that a static executive cannot [11]. The challenge, however, is that the executive 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.

The executives generally employ two primary strategies to efficiently reason about possible choices at run-time. First, 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* [13]. 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. Tsamardinos follows a similar strategy to handle disjunctive choices, expanding all the possible choices at compile time and determining the implications of any choices the executive might make. Second, Kirk uses an incremental strategy to efficiently explore the perturbations to

the plan induced by any particular choice at run-time. Chaski uses a hybrid approach, making pre-compilation more efficient by exploring the options with incremental reasoning.

This work develops Drake, a novel plan executive for plans with choice and uncertainty. Essentially, Drake frames the scheduling problem in the plan as a collection of Simple Temporal Problems differentiated by choices, and uses a pre-compilation strategy, reducing the problem to a dispatchable form. A dispatchable form is defined as a representation where scheduling decisions can be accurately made with only one-step constraint propagations at run-time. Once in this form, the plan is executed by a dispatcher. Drake's primary innovation is the use of techniques derived from the Assumption-based Truth Maintenance System (ATMS) to compactly encode the plan and perform reasoning at compile-time and run-time. Drake uses these techniques to delay the scheduling of events and the selection of discrete choices until run-time. Drake's compact encoding provides a reduction in the size of the plan used by the run-time executive by about four orders of magnitude for problems with around 10,000 component STPs, as compared to Tsamardinos's work. This compactness results in run-time latency that is two to three orders of magnitude slower for large problems, but is still tractable for real systems. These improvements are possible because the ATMS provided a straight-forward way to augment Muscettola's non-disjunctive STP algorithms to reason over choices in a compact way.

Section 1.1 provides an intuitive overview of Drake's techniques. Section 1.2 discusses related work.

1.1 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, below. Furthermore, it illustrates the compact representation that underlies this work and provides an intuition for why the representation is compact. For simplicity, we do not include uncertainty in this description.

1.1.1 Problem Statement

The Drake executive is designed to schedule temporal plans with a general notion of choice. This allows Drake to express important constructs for autonomous systems, including non-overlapping intervals, choices between sub-plans, and resources. Without choices, the executive necessarily has a single course of action it may follow. While temporal flexibility is useful, allowing discrete choices provides a new level of capability. For example, the executive may exchange possible sub-plans based on timing considerations. Another common use would be to indicate that two activities must occur, but cannot overlap. Without dynamic choices, the order of these two events would need to be set in advance, even if it is not useful to do so. Another important reason for disjunctions is handling resources, other non-temporal constraints on activities that may overlap. Drake does not currently include algorithms for determining the implications of resource choices, but the model is rich enough to express the temporal consequences of resource choices.

Input temporal plans are mapped into a Labeled-STP or STPU, which we introduce formally later, but which specify the constraints and choices of the plan in a compact way that is useful for the reasoning the executive performs. After a pre-processing step, Drake then executes the plan from the Labeled-STP representation.

Throughout this paper, we use the following simple example, which includes a choice between sub-plans.

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. □

The executive is responsible for executing the plan correctly, delaying decisions where possible. More specifically, it is responsible for selecting the times to schedule events, making decisions just before the events are actually scheduled. Additionally, it must instruct the physical system when to perform activities, and the durations the activities should take. For example, consider the following execution sequence for the rover problem. In this example, we neglect the decision-making strategy to focus on the outputs required from the executive.

Example 1.2 The executive begins the plan, arbitrarily denoting the begin time as $t = 0$. At that time, it instructs the system to begin the drive activity, indicating that the drive should take 40 minutes. The executive then waits until the system responds that that the drive has completed, at time $t = 45$. Then Drake selects the sample collection option, which had not been determined before, and initiates the activity with a duration of 50 minutes. At $t = 95$, the sample collection completes, finishing the plan within the time limit of 100 minutes. □

1.1.2 Labeled Distance Graphs and Compilation

For Simple Temporal Problems, creating a dispatchable form simply requires computing the All-Pairs Shortest Path graph of the distance graph associated with the input STP [13]. Adding discrete choices complicates compilation because the dispatchable form must then include the implications of both the temporal decisions and the discrete choices. In the rover example, the discrete choice is between collecting samples and charging the batteries. Additionally, the executive has flexibility over the precise start and end times of the activities. Each set of possible discrete choices implies a single *component* STP, which can be dispatched with standard STP techniques. A simple way to consider the consequences of the discrete choices, and the technique adopted by Tsamardinos, is to separately record the component STP for every combination of discrete choices [24]. This method is easily understood, but inefficient, because it assumes that every combination of choices is completely different from all others; this is rarely the case.

Instead, Drake introduces the Labeled-STP, a single, compact representation that contains all the constraints for all the choices. It exploits the fact that groups of choices may imply the same consequences, allowing Drake to collect a set of choices and treat them as a single entity during storage and reasoning [10]. These collections of choices are *environments*, and annotate, or "label," constraints within the Labeled-STP with the choices that imply the constraint [3]. Although there may be multiple possible constraints between any two events, the Labeled-STP allows them to coexist within a single representation because the environments unambiguously distinguish when each constraint is implied. This later strategy is leveraged from the ATMS [3].

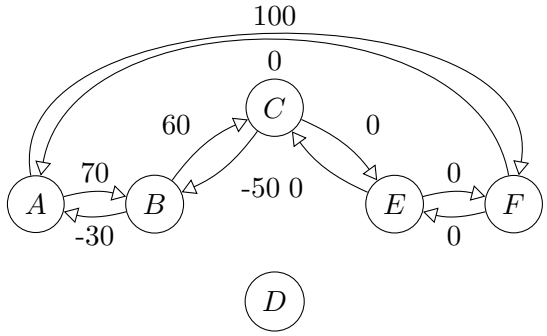
This labeled representation is especially powerful because it allows Drake to keep the essential form of an STP: a single network of events and constraints. It is then relatively straight-forward to generalize the non-disjunctive algorithm to handle disjunctive information.

Figure 1.1 shows how Tsamardinos's prior work independently records every constraint for every complete combination of choices [24]. 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.

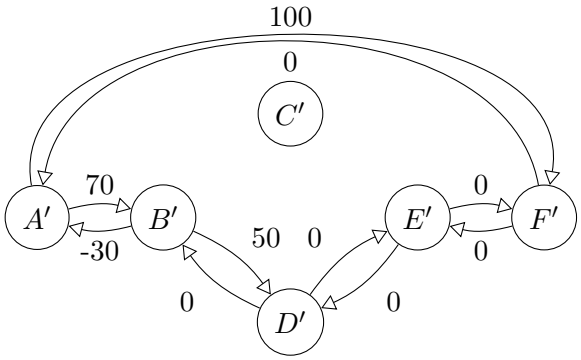
To reduce the repetition evident in this encoding, Drake builds a Labeled-STP representation of the choices and the constraints. This representation stores the values on edges in *labeled value sets*, applying the general framework of labeling and environments provided by the ATMS. In the rover scenario, we can record the constraint that the entire plan take less than 100 minutes only once. Where necessary, the implications of the choice between collecting samples and charging the batteries are given by constraints with environments. The resulting Labeled-STP is shown in distance graph form in Figure 1.2. 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. 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 1.1: The component STN distance graphs of the TPN in Figure 2.1.

(a) Collect Sample Component STN

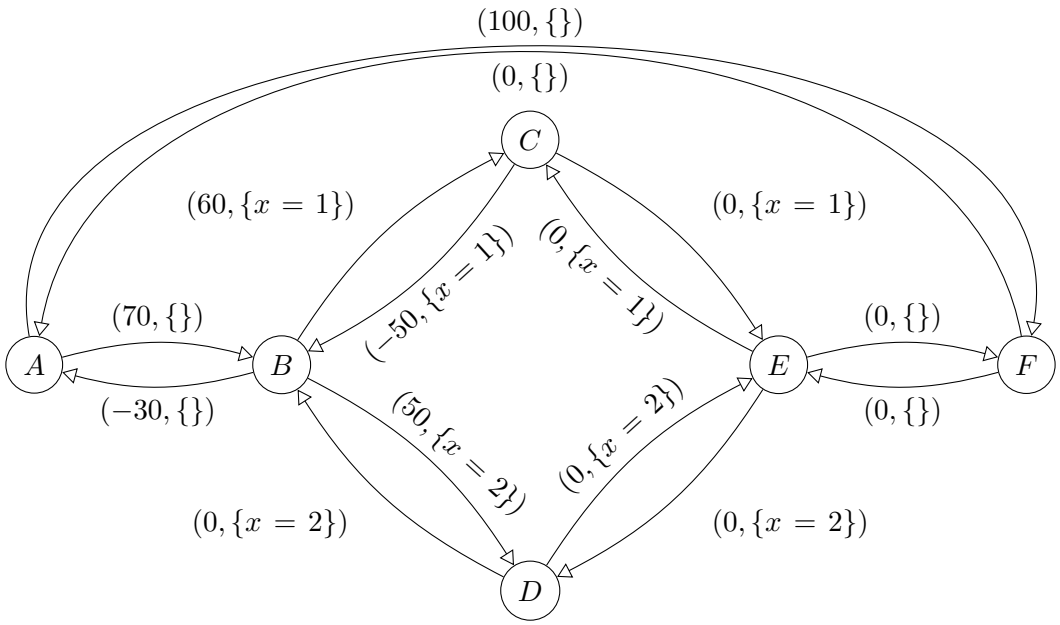


(b) Charge Component STN



After creating the Labeled-STP encoding of the plan, we need to compile it. In Tsamardinos's

Figure 1.2: The labeled distance graph corresponding to Example 1.1. All edges not drawn are implicitly valued $(\infty, \{\})$. The variable x denotes the choice and has domain $\{1, 2\}$.



approach, each explicitly enumerated STP is independently compiled. Drake directly performs an equivalent compilation on the Labeled-STP. For example, it uses edge (A, F), with weight $(100, \{\})$, and the collecting samples activity on edge (C, B), $(-50, \{x = 1\})$ to derive the restriction that the drive duration may only be up to 50 minutes if sample collection is selected. These two constraints have different environments, so the union of the environments is placed on the new constraint, $(50, \{x = 1\})$, stored as a weight on edge (A, B) (not drawn). There are now two constraints between events (A, B), which is necessary because the new constraint is tighter than the old one, but is only implied by some choices of the other constraint; thus, both constraints are useful. Specifically, the sample collection option implies a weight of 50, and the charging option implies a weight of 70. Generally, Drake would remove any constraints that are not useful. The result of the compilation is a dispatchable Labeled-STP , which directly records all the constraints the dispatcher needs to obey at run-time, as described in the next sub-section.

1.1.3 Dispatching the Dispatchable Labeled-STP

Dispatching using Labeled-STPs requires updating the non-disjunctive STP dispatching algorithm so that it handles the labels; this is a straightforward process. The following example demonstrates a few steps of Drake's dispatching process on this example in order to demonstrate the difference.

Example 1.3 Assume that the start event, A, in Figure 1.2, is scheduled at $t = 0$, the beginning of dispatching the plan. At some later time, $t = 40$, the executive needs to determine if it should schedule an event. Next, Drake considers scheduling the immediate successors of event A. An event cannot be scheduled until its predecessors have been scheduled, and must be assigned a

value consistent with the temporal constraints. Event A is B's only predecessor, hence B may be scheduled when the interval constraints are satisfied. The other events have B as a predecessor and must wait for it to execute. Thus, at time $t = 40$, Drake considers executing B. This time satisfies both constraints on B's execution, given by edges (A, B), $(70, \{\})$ and $(50, \{x = 1\})$ (not drawn). Therefore, B can be executed at $t = 40$ without any consideration of whether the rover will collect samples or charge. More generally, the time an event is scheduled must lie within the interval constraint for some environment. If the event is scheduled outside of some possible interval constraints, the executive may need to remove some choices from consideration. However, in this case, scheduling B at $t = 40$ is consistent with all possible options, so Drake maintains all its future options.

For example, if Drake repeated the same decision process at $t = 60$, it would notice that the constraint for collecting samples was violated, because $60 > (50, \{x = 1\})$. 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. □

1.2 Related Work

As a dynamic executive, Drake is derived from prior research on dynamic execution of TPNs and DTPs. The STP and DTP literature provides the underlying framework for dynamic execution [4, 13, 24]. Further work has developed more efficient methods for compiling STPs [15].

There are numerous extensions of the STP literature to create more capable scheduling frameworks. Most importantly, STPUs include a model of uncertainty which the executive can compile or dispatch in polynomial time [11]. Venable and Yorke-Smith added uncertainty to DTPs [25]. Tsamardinos introduced a probabilistic formulation of STPs [21]. Muscettola developed a technique for computing the impacts of resources on a temporal plan [12]. Khatib, Morris, Morris, and Rossi introduced a formulation including preferences for temporal scheduling decisions [8].

Kirk is a dynamic executive for TPNs [9]. Kirk performs optimal method selection just before run-time, assigning the discrete choices and then dispatching the resulting component STP. If some outcome invalidates the STP that Kirk chose, then Kirk selects a new STP consistent with the execution thus far. Further research developed incremental techniques to allow Kirk to re-plan with lower latency [17, 1].

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 [16, 24]. Shah points out that the component STPs of real-world TCSPs often differ by only a few constraints, allowing a compact representation. They record all the component STPs by storing a single relaxed STP and maintaining a list of modifications to the relaxed STP that recover each of the original component STPs [19]. 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 improvements in performance [16]. Our work is partially inspired by this success, and we further explore the connection to the ATMS.

The rest of this paper is organized as follows. Section 2 reviews the plan specifications and scheduling frameworks Drake uses. Section 3 develops our compact representation and the compilation algorithm for the deterministic case. Section 4 develops the dispatching algorithm for the deterministic case. Section 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 Section 6 and some conclusions in Section 7.

The appendices provide additional detail on some topics. Appendix A describes the conversion algorithm from TPNs and DTPs into labeled distance graphs. Appendix B describes the structured, random problem generators used in the experimental validation. Appendix C provides more detailed algorithms for the uncontrollable problem algorithms. Finally, Appendix D contains proofs of theorems presented throughout the text.

2 Temporal and Plan Representations

Drake builds upon prior work in plan representation for temporal reasoning and dispatchable execution. This section briefly provides some formal definitions and introduces the reader to the terms used in the literature. More details of the prior work are given in later sections, as needed.

First, Section 2.1 discusses Simple Temporal Problems, the underlying scheduling framework for this work. Second, Section 2.2 introduces Disjunctive Temporal Problems, one of the input formats employed by Drake. Finally, Section 2.3 defines Temporal Plan Networks, the other input format.

2.1 Simple Temporal Problems

Simple Temporal Problems provide a framework for efficiently reasoning about a limited form of temporal constraints and are the basis of the dynamic execution literature our work builds upon. A simple temporal network is defined as a set of events and temporal constraints among them [4].

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. □

The time of execution of these events is constrained through a collection of pairwise simple interval constraints.

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 $[l, u]$. □

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 not a strict ordering. Positive or negative infinities may be used in the bounds to represent an unconstrained relationship. Now we can define STPs.

Definition 2.3 (Simple Temporal Problem) The Simple Temporal Problem (STP) is a set of events V and a collection of simple interval constraints constraints between them, with exactly one constraint per pair of events. A *solution* is a scheduling of the events that satisfies all the the simple interval constraints. If and only if at least one solution exists, the STP is *consistent*. □

Dechter showed that STP reasoning can be reformulated as shortest path problems on an associated weighted distance graph [4].

Definition 2.4 (Distance Graph of an STP) A distance graph associated with an STP is a pair $\langle V, W \rangle$ of vertices V and weights W. The vertices exactly correspond to the events of the STP. The weights are a map $V \times V \rightarrow \mathbb{R}$ where the directed edge (A, B) has weight w_{AB} representing the inequality from the STP $B - A \leq w_{AB}$. □

Informally, computing the distance graph creates two edges for each simple interval constraint: one in the forward direction with weight u and one in the reverse direction with weight $-l$. Following Dechter's work, Drake primarily works with the distance graph form of the Labeled-STP.

Dechter proved that an STP is consistent if and only if its associated distance graph has no negative cycles, since such cycles correspond 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. We focus on the Floyd-Warshall APSP algorithm in Section 3.5 because it is a necessary component of the dispatching process [4, 13].

Muscettola showed that the APSP form of the STP's associated distance graph is a *dispatchable form*, where all constraints implicit in the original problem are made explicit, so that the network can be dispatched with only one-step propagations [13]. Since the run-time algorithm avoids inference over the entire graph, it is fast enough to use at run-time.

During execution, the dispatcher tracks *execution windows* that summarize the constraints on each event, which are updated by 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*. 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 [13].

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. These redundant edges may be trimmed, resulting in a *minimal dispatchable network* [13].

2.2 Disjunctive Temporal Problems

A formalism that directly modifies STPs to allow discrete choices is the Disjunctive Temporal Problem (DTP). We use DTPs as one of the input specification for Drake.

Definition 2.5 (Disjunctive Temporal Problem) A Disjunctive Temporal Problem is a collection of events V and a collection of disjunctive constraints. Each disjunctive constraint C_i is of the form

$$c_{i1} \vee c_{i2} \vee ... \vee c_{in}, \tag{1}$$

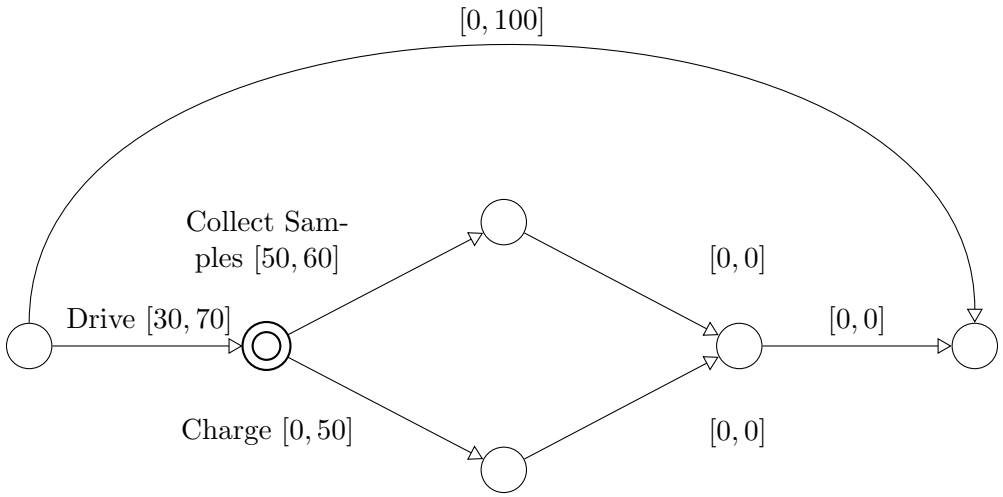
where n may be any integer and each c_{ij} is a simple interval constraint. There may be more than one simple interval constraint between any given pair of events. A *solution* to a DTP is an assignment to each event V so that for each C_i, at least one simple interval constraint c_{ij} is satisfied. A DTP is *consistent* if and only if at least one solution exists. □

The disjunctions expand the language of constraints expressible in STPs, allowing new concepts to be expressed, e.g. 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.

Most modern approaches for determining consistency of 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 [20, 14, 23]. 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, testing the consistency of a DTP can be understood as searching through the possible combinations of disjuncts for a consistent component STP.

Tsamardinos presented a flexible dispatcher for DTPs, which first enumerates all consistent component STPs and then uses them in parallel for decision making [24]. 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

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.



STPs, thus removing all possible choices from the DTP. Drake inherits this strategy for selecting between choices.

2.3 Temporal Plan Networks

A Temporal Plan Network is a graphical representation for contingent temporal plans introduced by Kim, Williams, and Abramson [9]. The primitive element in a TPN is an activity, comprised of two events connected by a simple interval constraint and an executable activity description.

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 included to conform to the hierarchical structure of a TPN.

The next section builds upon the work reviewed in this chapter to develop Drake's compact representation and compilation algorithms.

3 Compilation of Plans with Choice

Recall that Drake is a dynamic scheduling algorithm that can be employed to execute Temporal Plan Networks (TPN) or schedule Disjunctive Temporal Problems (DTP). In both these cases, Drake dynamically schedules events and chooses between alternatives, for example, disjunctive constraints or choice nodes. Drake does this efficiently by converting either TPNs or DTPs into a Labeled-STP, and then compiling it into a dispatchable form off-line. This section defines Labeled-STPs, distance graphs, and presents an algorithm for compilation. The new compilation algorithm adapts Muscettola's STP compilation algorithm to Labeled-STPs by incorporating environment propagation algorithms from the ATMS [3].

To provide a high-level overview of the compilation algorithm we begin with the top-level pseudo-code, presented in Algorithm 3.1. The input to the algorithm is a DTP or TPN, converted to a Labeled-STP and represented internally as a labeled distance graph described by events V, edges W, and variables X. The output of the algorithm is either a reformulation this graph into a minimal dispatchable form or a signal that the plan is infeasible. First, Line 2 prepares a data structure to hold the *conflicts* of the labeled distance graph, which are environments used to specify inconsistent choice. Second, Line 3 compiles the distance graph into dispatchable form with the LABELED-FLOYD-WARSHALL algorithm, revealing all implicit constraints of the problem. Finally if there are still some consistent component STPs, Line 7 filters the dispatchable graph of unnecessary edges. This section carefully defines both of these phases of the compilation process, which are directly taken from Muscettola [13]. We begin with a discussion of environments and the associated algorithms, then develop the compilation algorithms.

Algorithm 3.1 Compilation algorithm for Labeled Distance Graphs

```
1: procedure COMPILE(V, W, X)
2:     S ← INITCONFLICTDATABASE(X)
3:     W, S ← LABELED-FLOYD-WARSHALL(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
```

3.1 Introduction to Value Sets and Labeling

We observe that for real problems, the edge weights of the distance graph encoding each component STP are typically not unique for each set of choices. Rather, the values are loosely coupled to the choices selected for the plan. Therefore, we can avoid the explicit representation of each complete component STP and condense the possible weights for each edge into a single structure 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 single numeric weight. We could reconstruct the component STPs from the Labeled-STPs if necessary, but this is typically not helpful. 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 implications of any discrete choices in the plan.

The labeled value sets are made compact by introducing ideas from the Assumption-based Truth Maintenance System (ATMS) [3]. Each numeric value is labeled with an environment, as developed by the ATMS, 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 there are variables X and Y with domains $\{1, 2\}$, and some value depends only on X, we can represent the value with two entries, one for each value of X, rather than four entries, one for each pair of values X and Y might have.

Our implementation of labeled value sets 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.

3.2 Defining Labeled-STPs and Labeled Distance Graphs

This section presents an encoding for the choices contained in the input problem and defines the Labeled-STP and the *labeled distance graph*, the top level representations used in Drake's compilation technique. The following section then defines environments, which are used by the Labeled-STP and distance graph.

Both types of input problems of interest, TPNs and DTPs, are defined in terms of choices among temporal constraints. Drake uses *choice variables* to encode the choices.

Definition 3.1 (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 and includes a description of the domains. □

The Labeled-STP represents the scheduling problem and any choices in the input problem.

Definition 3.2 (Labeled-STP) A Labeled-STP is a set of events, V and a collection of constraints. Each constraint is a simple interval constraint between some pair of events of V, labeled with an environment. The environment is constructed of assignments to the choice variables described in X (see Definition 3.4). □

Note that there may be an arbitrary number of labeled constraints between pairs of events.

We now define labeled distance graphs, which Drake uses to store Labeled-STPs and on which we define Drake's algorithms. The representation essentially consists of the weights of the labeled constraints in distance graph form.

Definition 3.3 (Labeled Distance Graph) A labeled, weighted distance graph G is a triple $\langle V, W, X \rangle$. V is a list of vertices and W is a set of labeled value sets (see Definition 3.11) with domination function $f(a, a') := (a < a')$ (see Definition 3.9). The labeled value sets represent the weight function that maps the ordered vertex pair and an environment (see Definition 3.4) to weights: $V \times V \times \mathcal{E} \rightarrow \mathbb{R}$, for any vertex pair $(i, j) \in V \times V$ and environment $e \in \mathcal{E}$. X is the

description of the choices variables. The set of edges E contains those pairs of events (i, j) where $w(i, j) \neq \infty$, for some environment. All the labeled value sets in W are initialized with the pair $(\infty, \{\})$. □

Similarly to the unlabeled versions, weights in a labeled distance graph represent inequalities in the Labeled-STP. Only the tightest inequalities contribute to the reasoning (e.g. if $x \leq 5 \leq 6$, the six contributes no useful information), hence the domination function f is the less-than inequality. 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.

Converting input TPNs and DTPs into Labeled-STPs is a relatively simple process, requiring the annotation of constraints with environments that specify when they hold. The details are left to Appendix A. Converting a Labeled-STP into a labeled distance graph, as in the unlabeled case, requires converting the upper bound of the interval constraints into forward weighted edges and the negative of the lower bound of the interval constraints into backward weighted edges.

3.3 Environments and Conflicts

This section defines environments and describes the essential operations performed on them. Drake uses environments to specify the minimal set of choices that imply a constraint. They are also used in the process of deriving the dependence of new constraints on the choices. 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 [3]. Additionally, we discuss the concept of conflicts, which compactly record infeasible solutions, and describe the conflict manipulation routines Drake uses.

Given a set of choice variables and their corresponding domains, an environment is an assignment to a subset of the choice variables that summarizes the sufficient conditions for some derivation or computation to hold. Drake builds environments exclusively with assignments to choice variables. Consistent environments assign at most one value to each variable.

Definition 3.4 (Environment) Given a set of variables X and their domains, an *environment* e is a list of assignments to a subset of the variables in X, written $e = \{x_i = d_{ij}, ...\}$. An environment must have at most one assignment to each variable to be consistent. Thus, an environment of the form $\{x_i = d_{ij}, x_i = d_{ij'}, ...\}$ is always inconsistent if d_{ij} and $d_{ij'}$ are distinct. 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 environment as $\mathcal{E}_c$. The length of an environment is the number of assigned variables, denoted $|e|$. □

In the ATMS, a proposition may be labeled by a set of environments, where each environment logically entails that proposition [3]. For example, in labeled distance graphs each edge weight value w corresponds to a proposition $x - y \leq w$, where x and y are the events the edge connects. Drake gives each proposition exactly one environment, but the proposition may occur multiple times with different environments. 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. Therefore, reintroducing unique propositions into 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 entails another. Union is the primary way to combine environments to create new ones for logical consequences of propositions

Definition 3.5 (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 $\{x_i = d_{ij}\} \in e'$. □

Example 3.6 An environment $\{x = 1, y = 2, z = 1\}$ is subsumed by $\{x = 1, z = 1\}$ because the assignments in the later environment are all included in the former. □

Subsumption is generally used in practice to determine whether a proposition applies in some scenario or if one labeled proposition is redundant to another one.

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

Definition 3.7 (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 union and $\perp$ is returned instead. This value signifies that there is no environment where both e and e' hold simultaneously. □

Example 3.8 Most commonly, unions are used to compute the dependence of new derived values. If $A = 2$ when $\{x = 1\}$ and $B = 3$ when $\{y = 2\}$, then $C = A + B = 5$ when $\{x = 1\} \cup \{y = 2\} = \{x = 1, y = 2\}$. □

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 choices are still possible. The standard strategy in an ATMS is to keep a list of minimal *conflicts*, also referred to as *no-goods* [3].

A *conflict* is an environment that entails an inconsistency [27]. For example, the compilation process might determine that $x_1 = 1$ and $x_2 = 1$ are contradictory choices, and cannot be selected together during any execution. Then, $\{x_1 = 1, x_2 = 1\}$ is a conflict of the system. By definition, all environments subsumed by this conflict also contain the inconsistency and are invalid. Therefore, conflicts are used to summarize the inconsistent environments.

The other important function of the conflict database is to determine if all the complete environments have been invalidated, that is, determined inconsistent, or if a certain conflict would invalidate all environments. 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 x_1 is feasible. Therefore, the entire Labeled-STP is inconsistent. During compilation and dispatch, Drake keeps track of what choices have not been invalidated as part of the reasoning.

Our algorithms use a database of conflicts to determine if an environment is known to have been invalidated. In our pseudo-code, we call the conflict database data structure S. This database keeps a set of minimal conflicts and is updated as conflicts are added. We now define the functions Drake uses to interact with the conflict database, but leave details of the algorithms required to Williams et al. [27].

The two fundamental operations in Drake are testing an environment for consistency and adding conflicts. EnvironmentValid?(S, e) tests whether a given environment is known to be inconsistent given the current conflicts. AddConflicts(S, e) adds the environment e as a conflict. It

returns true if all possible environments are inconsistent after adding the conflict, otherwise it returns false.

The function EnvironmentsRemain?(S) returns true if any complete environments have not been invalidated. ConflictsPossible?(S, l) queries the database, returning true if making conflicts from the environments in list l would not invalidate all the complete environments. This function allows Drake to avoid eliminating all its remaining options. 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 to invalidate non-subsumed environments.

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 and to manipulate those dependencies during reasoning. With this formalism defined, we can explain the implementation of labeled value sets.

3.4 Labeled Value Sets

This section defines labeled value sets and describes Drake's implementation of them. We describe this implementation carefully before moving on to the compilation algorithm because the compact representation is the core contribution of this work. The purpose of labeled value sets is to allow Drake to compactly map from choices in the input plan to constraints implied by those choices. 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 that are 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 to use on a given labeled value set. We say that a tighter value *dominates* a looser value.

Definition 3.9 (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', exactly one of $f(a, a')$ or $f(a', a)$ must return true. □

Drake uses strict inequalities for the domination function, which provide a total ordering over all real numbers.

Example 3.10 The most important use of domination functions is determining whether a value is redundant. If we derive two potential values for an edge weight, say, 3 or 5, the domination

function reveals which one we need to keep. Edge weights are ordered with $f(a, a') = a < a'$, so $f(3, 5) = \top$. Thus, the 3 is the value we need to keep. □

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

Definition 3.11 (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. □

Example 3.12 A variable that has value 3 when $x = 1$, and is 5 otherwise, is represented by the labeled value set $\{(5, \{\}), (3, \{x = 1\})\}$. □

Drake interacts with labeled value sets by querying for a value, adding a value, and by performing binary operations on two labeled value sets. 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 may be returned if its environment subsumes the input environment. Of the possible values, the dominant value is returned. Formally:

Definition 3.13 (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 $f(a_i, a_j) = \top$, for all e_j present in any pair $(a_j, e_j) \in A$ 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 and remove any newly redundant entires.

Definition 3.14 (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 $A \leftarrow A \cup (a, e)$. We also prune any values from A that are redundant, meaning that no query can return them. □

After new values are added to the labeled value set, the set may not be compact because the set might contain redundant values. The following example illustrates how the structure of domination and subsumption is used to 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 3.15 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 we 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 values to return from the query. However, three

dominates five, and is therefore returned. Now imagine that we add the labeled value $(2, \{x_1 = 1\})$. Similarly, the new pair is added to the set, resulting in

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

Notice that $A(e)$ would 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. Hence, we remove the value three and its environment.□

It is guaranteed that this form answers queries with no loss of information and prove the correctness of the uniqueness criteria.

Theorem 3.16 (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* $A(e)$. □

Algorithm 3.2 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 Theorem 3.16. The input to the function AddCandidateValues 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 updated labeled value set, which may have new values inserted, and if so, may be pruned of some old values that the new values make unnecessary. The outer loop simply processes each value of the new value set B.

Algorithm 3.2 Add new elements to a labeled value set, maintaining minimality.

```
1:  procedure AddCandidateValues(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 subsumed
4:              if (e_j subsumes e_i)&&(f(a_j, b_i) == ⊤) then
5:                  continue A                  ▷ Not subsumed, 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(b_i, a_j) == ⊤) then
10:                 A ← A \ (a_j, e_j)                 ▷ Old value is subsumed, prune
11:             end if
12:         end for
13:         A ← A ∪ (b_i, e_i)
14:     end for
15:     return A
16: end procedure
```

To illustrate this algorithm, reconsider the last step of Example 3.15. 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 is not subsumed, 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 of pair $(3, \{x_1 = 1\})$, 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 3.15.

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

We conclude by defining arbitrary binary operations on 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 Section 5 uses it to apply several different propagation rules. First, we show a technique for performing operations on individual pairs of values with environments taken from [3].

Lemma 3.17 (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 $(g(a, b), e_a \cup e_b)$. □

Applying binary operations to entire labeled value sets requires taking the cross product of the input sets. This is justified by Theorem 3.18.

Theorem 3.18 (Binary Operations on Labeled Values) *For two labeled value sets A and B, 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.* □

To perform a binary operation g on labeled value sets, 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 3.3 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:

Algorithm 3.3 Calculate the results of a binary operation on a minimal dominant labeled value set.

```
 1: procedure LabeledBinaryOp(A, B, f, g, S)              ▷ Compute C ← g(A, B)
 2:     C ← {}
 3:     for (a_i, e_i) ∈ A do
 4:         for (b_j, e_j) ∈ B do
 5:             c_ij = g(a_i, b_j)                        ▷ Calculate the candidate
 6:             e_ij = e_i ∪ e_j
 7:             if (e_ij ≠ ⊥) ∧ EnvironmentValid?(e_ij) then   ▷ Keep if valid. Sec 3.3
 8:                 C ← AddCandidateValues(C, {(c_ij, e_ij)}, f)   ▷ Alg. 3.2
 9:             end if
10:         end for
11:     end for
12:     return C
13: end procedure
```

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

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, inconsistent environments are also removed, so $(5, \bot)$ 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, discarding some unnecessary values. Therefore, the final result is:

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

This section has defined labeled value sets, a compact representation for values that depend on assignments to discrete values. The operations we have defined here allow us to simply integrate this data structure into existing algorithms.

3.5 Labeled All-Pairs Shortest Path

Next, we consider how to compute the dispatchable form of a labeled distance graph. The implicit constraints between events are exposed by applying a variant of the Floyd-Warshall All-Pairs Shortest Path algorithm, which is developed in this section [2]. The input labeled distance graph is a compact representation of all the initial component STPs, hence a single run of Labeled-Floyd-Warshall compactly computes all the compiled component STPs, replacing Tsamardinos's approach of compiling 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.

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 component 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 possible choices are invalidated. Additionally, Drake avoids performing computations that are only relevant to choices that are known to be inconsistent, as is standard for improving efficiency in an ATMS [3].

Algorithm 3.4 Labeled APSP Algorithm

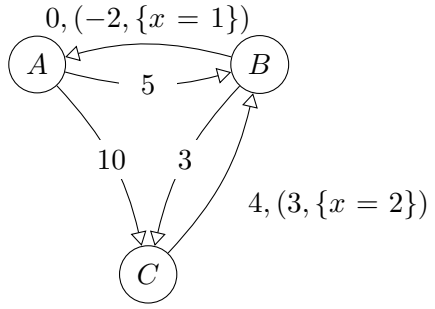
```
1:  procedure LABELED-FLOYD-WARSHALL(V, W, S)
2:      for i ∈ V do                                   ▷ Cycle through triangles
3:          for j, k ∈ V do
4:              C_jk ← W_ji + W_ik                     ▷ Apply "+" with Alg. 3.3
5:              if j == k then                         ▷ Self-loop update
6:                  S ← CHECKFORNEGCYCLES(C_jk, S)
7:              else                                   ▷ Non-self-loop update
8:                  W_jk ← ADDCANDIDATEVALUES(W_jk, C_jk, '<')   ▷ Alg. 3.2
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:         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 3.4. Recall that the Floyd-Warshall algorithm updates the shortest paths by looking for a route $j \to i \to k$ that provides a smaller weight than the weight on the existing edge $j \to k$. The Labeled-Floyd-Warshall algorithm is nearly identical to the standard Floyd-Warshall algorithm [2]. 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, edges $i \to i$, are not stored, but are checked for negative cycles to find inconsistent component STPs. Third, the non-self-loop candidate edge weights are added to existing labeled value sets with the AddCandidateValues operation.

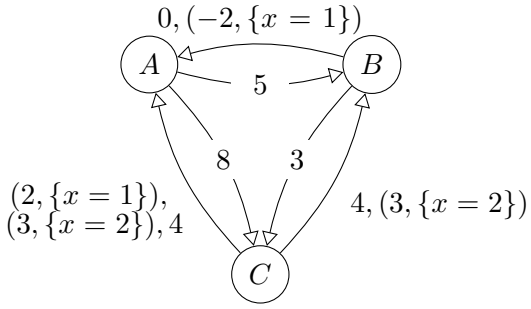
We can illustrate this algorithm by compiling the small distance graph in Figure 3.1 with a single choice $x \in \{1, 2\}$. Stepping through each step of the Floyd-Warshall algorithm is tedious for even three nodes, so we only present selected steps. The outer for-loops iterate through triangles of the graph, deriving shorter path lengths. Line 4 computes the path lengths that two sides of

Figure 3.1: A simple example of running LABELED-FLOYD-WARSHALL. Unlabeled values have an implicit empty environment. For example, 10 represents $(10, \{\})$

(a) Input labeled distance graph



(b) Output APSP labeled distance graph



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})
$(4, \{\})$	$(0, \{\})$	$(4, \{\})$
$(3, \{x = 2\})$	$(0, \{\})$	$(3, \{x = 2\})$
$(4, \{\})$	$(-2, \{x = 1\})$	$(2, \{x = 1\})$
$(3, \{x = 2\})$	$(-2, \{x = 1\})$	$(1, \perp)$

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 input environments give competing values for x and their union is therefore inconsistent. 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 executive will eventually select a value for x, and thereafter either 2 or 3 is returned in response to a query. Since both values of x have dominating 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. This is future work, and can be handled with a multi-resolution rule.

No further propagations update any of the labeled value sets and the updated graph is shown in Figure 3.1. To illustrate the self-loop update, consider computing the self-loops for C created by following the path to B. This path induces weights $(7, \{\})$ 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 searches every labeled value set and removes every labeled value whose environment is subsumed by the new conflict. This process avoids 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 $\mathrm{O}(N^3)$ of Floyd Warshall.

Theorem 3.19 *The* Labeled-Floyd-Warshall *function shown in Algorithm 3.4 produces a labeled representation of the APSP of all the consistent component STPs of the input DTP.* □

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.

3.6 Pruning the Labeled Distance Graph

Muscettola et al. developed a post-processing step for dispatchable networks to prune redundant edges [13]. 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 [13].

Simply put, Muscettola proves that an edge is dominated, and may be removed, whenever its weight is exactly propagated by the other two sides of a triangle in the graph.

We generalize the prior work to the following theorem governing domination in the presence of environments.

Theorem 3.20 (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* $(w_k^{BC}, e_k^{BC}) \in W(B,C)$*.*

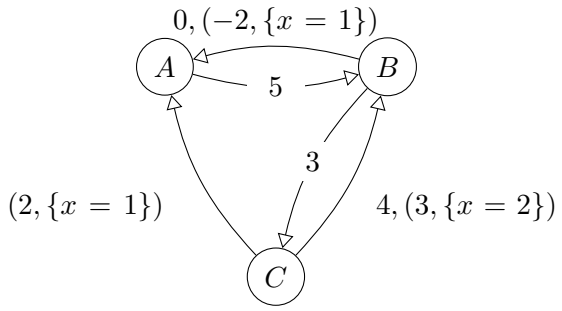
1. *A non-negative, non-zero edge weight* w_i^{AC} *is upper-dominated by another positive, non-zero edge weight* w_k^{BC} *if and only if* $w_i^{AB} + w_k^{BC} = w_j^{AC}$ *and* $(e_i^{AB} \cup e_k^{BC})$ subsumes e_j^{AC}
2. *A negative, non-zero edge weight* w_i^{AC} *is lower-dominated by another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

Intuitively, $A \to C$ is upper dominated and may be removed if it has a positive weight, which 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 after running Labeled-Floyd-Warshall, the theorem's hypothesis is satisfied and the domination test for edge pruning applies.

Muscettola also provides a search algorithm which identifies all possible prunings with a marking scheme, before removing any edges, which ensures that maximal pruning occurs [13]. This search algorithm is suitable without modification, so we do not repeat it here.

Example 3.21 We illustrate the filtering algorithm by continuing from the APSP labeled distance graph shown in Figure 3.1. For space considerations, 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 3.2. □

Figure 3.2: The filtered DTP from Example 3.21.



Note that there is another, more complex algorithm for STPs, which interleaves the APSP computation and edge filtering. This algorithm avoids the expansion and contraction that is characteristic of the APSP and filtering process and provides a lower run-time bound [22]. This other algorithm could likely be modified with our labeling technique in future work. This filtering process completes our development of a compilation algorithm for labeled distance graphs.

4 Dispatching Plans with Choice

In this section, we describe how to take the original STP dispatcher developed by Muscettola, Morris and Tsamardinos, and update it to work with labeled value sets and labeled distance graphs [13], yielding Drake's dispatching algorithm for deterministic problems. This dispatcher uses labeled value sets to compactly encode both the compiled problem and the bounds computed at run-time. This chapter also modifies Muscettola's STP dispatcher to schedule *activities*, which are components of TPNs used to represent some real-world processes [9]. 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.

Tsamardinos performs dispatching on a DTP by maintaining every compiled STP in memory and by updating them all in parallel, explicitly recording the consequences of the choices. Thereby, the executive can begin the execution with all its options available, and incrementally select between them as the execution unfolds. Drake adopts this broad strategy, but implements it more efficiently by using labeled distance graphs and value sets.

This section presents Drake's dispatching algorithm for deterministic problems. This algorithm takes as input a dispatchable labeled distance graph, and dynamically executes it, assigning times and choices on the fly. At the end of execution, Drake will have selected a single component STP from the labeled distance graph whose constraints are satisfied, but avoids committing to a particular STP sooner 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 by first reviewing standard STP dispatching, adapting these techniques to handle labels, and then describing the modifications necessary for activities.

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 [13]. This top level algorithm is quite simple. Essentially, it loops, searching for events to schedule, until either every event is scheduled or a failure is detected. Determining if an event is schedulable only requires determining if the constraints between the event and its neighbors are satisfied, which is performed in two steps: testing that the inequalities encoded in the distance graph are satisfied and testing that the ordering constraints are satisfied.

The dispatcher efficiently tracks the times when an event may be executed by computing *execution windows* for each event. Execution windows are the tightest upper and lower bounds derived for each event through the one-step propagation of execution times. Checking that the current time is within an event's execution window is sufficient to ensure the temporal constraints encoded in the distance graph are satisfied, if the ordering constraints are also satisfied.

Testing whether the predecessors of an event have been executed is called testing for *enablement*. A simple temporal constraint may imply a strict ordering between two events, which the dispatcher must explicitly test to ensure that an event is not scheduled before an event that must precede it.

Drake's dispatcher relies on these two fundamental tests, developed by Muscettola, for whether an event is executable. Drake adds support for storing the dispatchable graph in a labeled distance graph and stores the execution windows in labeled value sets. Drake also adds reasoning steps to allow it to consider the possible choices available and to select between them. Finally, it adds support to find and execute activities.

Algorithm 4.1 provides pseudo-code for Drake's top level dispatcher, called DispatchLabeledDGraph. The dispatcher uses functions developed throughout this chapter. The input to the dispatcher is the dispatchable version of the input problem, and is specified by the events V, the labeled value sets representing the edge weights, W, and the conflict database, S. It also takes a representation of the activities, Act, which is defined fully in Section 4.5. The activity structure specifies which intervals of the original plan are activities, their environments, and some identifier of what physical activity it represents. The result of dispatching is that either the events and activities are executed according to the constraints implied by one set of choices, 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 rover example, given in Example 1.1 and shown in Figure 1.2. 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. □

The first step of the algorithm, Line 2, initializes sets to hold the events that have been executed and events that are still waiting on activities to be executed. Also, the initial time is set to zero, without loss of generality, as is typical in the literature. Lines 4-7 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 executes it if that is the case. 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 for scheduling them. First, Line 24 calls EventExecutable?, developed in Section 4.3. At any time, executing a particular event at that time might be consistent with all the possible choices, none of the choices, or some of the choices, which the function determines through operations on the conflict database. 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 to the next possible event. 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 implied by the edge weight $(-30, \{\})$ is not met, yet is required for all possible choices.

If 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. If the scheduling decision is only consistent with some of the remaining choices, then conflicts are added to the database to represent those choices excluded by the time assigned to the current event. For example, event C must follow B in the rover example, if the rover collects samples. However, it can create a conflict for sample collection, $\{x = 1\}$, and then disregard the ordering constraint and schedule C before B.

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 using a propagation algorithm developed in Section 4.2.

Now the dispatcher executes 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 scheduled and are consistent with the choices available. In this case, the

Algorithm 4.1 The top level dispatching algorithm.

1: **procedure** DispatchLabeledDGraph($V, W, S, Act, \Delta t$)
2: $V_{exec}, V_{waiting} \leftarrow \emptyset$ ▷ Initialize event sets
3: $t \leftarrow 0$
4: **for** $i \in V$ **do** ▷ Initialize execution windows
5: $B_i^u \leftarrow (\infty, \{\})$
6: $B_i^l \leftarrow (-\infty, \{\})$
7: **end for**

8: $(B, S, V_{exec}, v) \leftarrow$ ExecuteIfPossible($V, W, V_{exec}, S, B, V_{start}, 0$) ▷ Execute start event
9: $V_{waiting} \leftarrow V_{waiting} \cup v$

10: **while** $V \neq V_{exec}$ **do**
11: $S \leftarrow$ CheckUpperBounds(V, W, V_{exec}, S, B, t) ▷ Alg. 4.4, find violated upper bounds
12: $V_{finished} \leftarrow$ GetFinishedActivities()
13: $V_{waiting} \leftarrow V_{waiting} \setminus V_{finished}$

14: **for** $i \in V \setminus V_{exec} \setminus V_{waiting}$ **do** ▷ Try to events
15: $(B, S, V_{exec}, v) \leftarrow$ ExecuteIfPossible($V, W, V_{exec}, S, B, i, t$)
16: $V_{waiting} \leftarrow V_{waiting} \cup v$ ▷ store starting activities
17: **end for**

18: $t \leftarrow t + \Delta t$ ▷ 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** $S_{removed} \leftarrow$ EventExecutable?$(V, W, V_{exec}, S, B, i, t)$ **then** ▷ Alg. 4.3
25: $B \leftarrow$ PropagateBounds($V, V_{exec}, W, S, B, i, t$) ▷ Alg. 4.2
26: $(S, V_{waiting}) \leftarrow$ BeginActivities($V, V_{exec}, W, S, B, i, t$) ▷ Alg. 4.5
27: $V_{exec} \leftarrow V_{exec} \cup i$ ▷ Store execution
28: **end if**
29: **return** $B, S, V_{exec}, V_{waiting}$
30: **end procedure**

dispatcher finds the drive activity, which must begin with the start event regardless of which choices the dispatcher makes. Drake tells the system to concurrently 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 concurrently, 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 drive activity is done. Later, Line 12 performs a system call that polls whether any activities are complete, signaled by returning B. Once this occurs, 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 scheduled. 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, meaning that the current time is beyond the upper bound of an execution window for the 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 possible choice. 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 whether the activity's environment has been invalidated. Therefore, while the events for unselected paths become unconstrained, since the dispatcher does not enforce constraints labeled with invalidated environments, 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 that activity because its environment is inconsistent. See Section 4.5 for a more detailed development of the activity reasoning algorithms. The remaining sections fill in the details of this top level algorithm.

4.2 Labeled Execution Windows

Drake requires a labeled analogue to execution windows, developed by Muscettola, to dispatch events. In an STP dispatcher, the execution windows are maintained as a single upper and lower bound on each event's execution time [13]. In a DTP, Tsamardinos computed and stored bounds independently for each component of STP [24]. 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 by propagating an execution time through an edge of a labeled distance graph, producing a labeled value set. This representation shows the dependence of the execution windows on the choices, which the dispatcher uses to determine 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, hence 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 $f(a, a') := (a > a')$ and B_i^u is a labeled value set with $f(a, a') := (a < a')$. 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 [13]. 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 dispatchable labeled distance graph from Example 3.21, computed in Section 3 and shown in Figure 3.2. 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 the 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 3.4 ensures that all of the bounds represented are useful for some possible execution, according to

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	$(\infty, \{\})$	$(\infty, \{\})$

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

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

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.

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. 3.2 and Alg. 3.3
5:         B_j^u ← MERGECANDIDATES(B_j^u, B_i^u + W_ij, S, <)      ▷ Alg. 3.2 and Alg. 3.3
6:     end for
7:     return B
8: end procedure
```

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

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.

4.3 Selecting Events to Execute

This section develops EventExecutable? in Algorithm 4.3, which is the function that Drake repeatedly calls to determine if a particular event is executable at the current time, given the dispatchable 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.

Drake can select between different possible choices 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 choices, down to one, or a few, that it actually executes and satisfies all its constraints. It is possible that Drake might reach the end of an execution having satisfied the constraints of several component STPs, having some choices that are unresolved, but we consider that a happy coincidence the system does not care about. To make these decisions while guaranteeing correctness, we use Tsamardinos's strategy in the following sense: Drake is allowed to execute an event at any time that is consistent with at least one of the remaining choices. After scheduling the event, the choices that are inconsistent with this schedule are removed from consideration by creating conflicts. If only one choice remains, Drake must follow it exactly. If, because of some external event or unexpected delay all remaining choices 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.

Example 4.6 Let us return to Example 4.3, where we consider a single event. There is one choice, x, with two possible options. If the executive selects $\{x = 1\}$, the event must 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, loosely speaking, dispatching this event requires the executive to make a decision between them. In practice, Drake will identify that it can schedule the event at some time, say, $t = 3$, and will schedule the event. It also notes that this decision violates the lower bound for the window corresponding to $\{x = 1\}$, thus invalidating the possibility of satisfying the constraints for that choice. Therefore, Drake records this invalidation 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 constraints 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. If Drake violates a constraint at run-time, the environments

of those violated constraints become conflicts. For instance, violating the window $[5, 10]$ from the example above requires creating the conflict $\{x = 1\}$, because this is the environment for the violated constraint. The conflict exactly summarizes which choices are invalidated by that execution. Instead of finding the choices 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 schedulable, then, requires collecting the environments of activation constraints and execution windows that would be violated by scheduling 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. 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 would 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. □

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 dispatchable labeled distance 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. Essentially, the algorithm finds all the conflicts it would need to create to schedule the input event at the current time, and then determines whether or not it can do so without invalidating all possible choices, before the system actually schedules the event. 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 if Drake went ahead with scheduling event i at time t. Line 2 initializes $e_{violated}$, a set that will hold the environments of violated constraints. Then, Lines 3-8 goes through through the upper bounds on the event i and stores the 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 subsequent 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 find any events that must be scheduled before the event under consideration, which have not yet been scheduled. It does so by looking for negative

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
```

outgoing weights from i to non-executed events, as these imply strict ordering constraints that are not currently met. The environments of any weights it finds are recorded. 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 previously executed, hence no constraint is violated.

Having collected all the environments of violated constraints, the second phase of the algorithm determines if these environments would invalidate all possible choices if they became conflicts. Line 20 queries the conflict database to perform this test. If that returns true, then the conflicts are created and the algorithm returns true, indicating that the event should be scheduled at the current time. Otherwise, the event cannot be scheduled at the current time, and the algorithm signals this by returning false. Since the event will not be scheduled at the current time, the environments collected are not conflicts, and are not added to the conflict database. 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, $\{x = 2\}$.

Empty environments correspond to constraints that apply universally, hence, making $\{\}$ a conflict necessarily invalidates all possible environments. Therefore, the dispatcher is never allowed to violate a constraint with an empty environment.

EventExecutable? 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? *indicates that the input event is executable if and only if it is executable in one of the consistent component STPs.* □

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 scheduling 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 scheduled in the future, Drake simply schedules events as soon as possible, in order to simplify the activity algorithm, discussed in Section 4.5. While we do not explore the possibility here, it should be possible mirror Tsamardinos's approach to reasoning about future execution times.

This section presented the algorithm for determining if an event may be scheduled at the current time and for pruning conflicting choices. It closely follows the strategies provided by Muscettola and Tsamardinos for performing the reasoning, while adapting the steps to our representations.

4.4 Finding Violated Bounds

When selecting events to execute, the dispatcher is allowed to directly violate the constraints within some component STPs as long as there exists some alternative choice where no constraints are violated. However, in a dynamic execution system, there may be unexpected delays that prevent the system from scheduling events when they should occur, violating constraints that Drake would

not intentionally disregard. Therefore, the dispatcher needs to routinely check whether the any such delays have occurred and note any violated constraints. As was the case for event selection, violated constraints induce conflicts to represent the invalid choices. However, since the dispatcher is not 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 checks for upper bounds that have been violated and prunes 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 if Drake completely controls all the scheduled durations, 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 networks with activities requires different temporal semantics than Simple Temporal Problems, because an activity's duration must be set when it begins. Within a temporal plan, activities are typically modeled with a start and an end event. When temporal plans are scheduled with an STP dispatcher, the dispatcher 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 general need to model activities, this section proposes a method for modifying an STP based dispatcher, such as Drake, to handle this case. Specifically, we develop a function BeginActivities, which is called when an event is scheduled, 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 executing activities under an STP dispatcher.

Example 4.10 The running rover example begins with a drive, which demonstrates the incompatibility of STP dispatchers and activities. If the drive ends with the rover at a pre-determined destination, the flexibility in the drive activity implies that the rover might be able to go faster or slower. Thus, the STP dispatcher may schedule the end event at any time between 30 and 70 minutes. However, to accomplish these feasible end times, the rover needs to select a duration at the beginning of the drive so it may select a drive speed appropriately. □

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 selects the duration of the activity at the end of the interval or whether it must do so at the beginning, respectively [26]. Later on, STP dispatchers adopted end controllable durations throughout, but many physical systems can only be reasonably modeled with begin controllable durations. Drake only allows begin controllable activities, but since end controllable semantics are exactly those used by STPs, it would be simple to schedule them without the advance duration selection we now propose.

Definition 4.11 (Activities) An activity in a temporal plan is a duration that has some activity the executive must execute. The activity connects a *start* and *end* event. Its duration may range from l to u, the lower and upper bounds, but the duration is set when the start event is scheduled. The activity may be labeled with some environment, e, representing the choices the dispatcher must make to execute this activity. Finally, there is some *primitive*, which describes the actual activity to execute. □

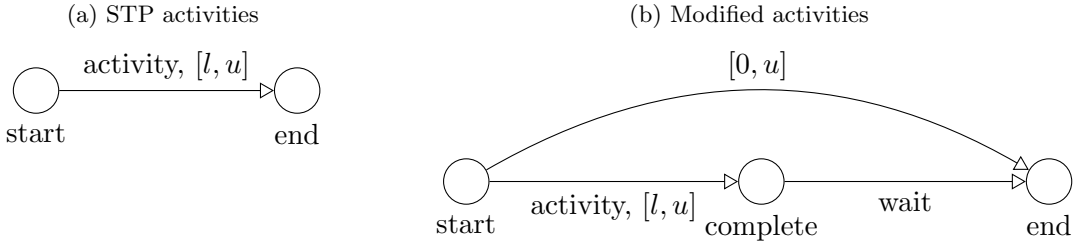
The activities of a temporal plan are summarized in the data structure *Act*, which is simply a list of the activities, where each element of the list has a field for each description of the activity given in the definition. Note that the simple interval constraint represented by the activity is also encoded directly into the Labeled-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.□

We need a way for Drake to dynamically select the durations of activities at their beginning, without assuming that the activity will complete in exactly the expected time. Therefore, we change the interpretation of activities to separate the end of the actual physical process from the end of the activity, as shown in Figure 4.1, in a strategy we refer to as "hurry up and wait." We insert a node between the start and end of the activity, which represents when the physical process completes. In this model, the dispatcher selects the duration of the activity at the start, and then adjusts the wait duration to accommodate any delays in the execution. 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. Timing propagations from other events cannot cause this strategy to fail, 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, which the wait allows the dispatcher to do. Note that Drake never performs this transformation on the distance graph, but implicitly assumes this transformation.

There are two important properties of this technique. First, it assumes that requiring the activity to idle is feasible; while this is reasonable in many cases, it is not universally applicable. Second, it cannot necessarily adapt to arbitrary delays in the system, instead it allows the system to make a good-faith effort to adjust to disturbances, which is an improvement over fixed schedules that fail with any disturbance whatsoever. If guarantees of success are needed, or significant disturbances are expected, an explicit model of the uncertainty, as described in Section 5, is necessary.

Figure 4.1: An illustration of Drake's interpretation of activities, the "hurry up and wait" model.



Definition 4.13 (Activity Execution) 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, so that X is the start event of the activity and precedes Y, the end event. The definition states that the activity duration must be within the $[l, u]$ interval, and also that the duration plus any wait inserted must lie within the $[l, u]$ interval.

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 schedule 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 activity's constraint. □

With this relaxed definition, we can prove the correctness of a 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, meaning that all choices that do not include the activity have been invalidated. This activity specifies 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 scheduled. If there is no current lower bound for Y, simply set $t_{exec} = l$.* □

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 dispatcher is committed to the activity, because otherwise, the temporal constraints might allow the dispatcher to reverse its decision to begin the activity, which we cannot allow. 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 begins an activity, it has naturally committed itself to any choices necessary for that activity.

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, Drake can minimize the wait time by starting the activity with length seven. □

This technique for selecting execution times is summarized in Algorithm 4.5. Its inputs are the dispatchable labeled distance graph, the summary of the activities, 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.

The algorithm determines if each activity should execute, and if so, how long it should take. Line 4 tests that the environment is still consistent. 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

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
```

on the end event, whichever is greater. This is computed on Lines 6 - 12. In the rover problem, the execution duration is 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 Drake only needs to 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, releasing 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. Here we assume that this activity is the only one ending at this event.

4.6 Conclusion

This section 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 section 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.

5 Plans with Choice and Uncertainty

Morris et al. introduced a dispatcher with a model of uncertainty that demonstrated that the great strength of compilation and dispatchable execution is its ability to provide explicit guarantees about whether the executive can successfully execute a plan, even if some of the durations are not controllable by the executive [11]. Furthermore, reasoning about set-bounded uncertainty in this model is possible in polynomial time. Specifically, this prior work extended Simple Temporal Problems to include bounded uncertain durations in the problem specification, creating Simple Temporal Plans with Uncertainty (STPU). The work also introduced a compilation algorithm that can analyze a problem in order to determine whether a dispatcher can execute the plan correctly, thus guaranteeing robustness to the modeled uncertainty. This section outlines techniques to replace Labeled-STPs with Labeled-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. Since most of the algorithmic insights are identical to those used in the previous sections, we focus on an overview of the approach and leave details to Appendix C.

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. 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 implying 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 resolves quickly, this option is actually feasible, and charging provides an acceptable backup if the drive is slow. However, this approximation allows us to develop the Labeled-STPU and leverage the work of the previous sections. A Labeled-STPU is defined similarly to a Labeled-STP, except that some edges are marked as uncontrollable. As before, all edges are labeled with environments.

This section outlines Drake's approach to reasoning 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 modify the existing algorithms to use labeled value sets. In this case, we modify Stedl's fast dynamic controllability algorithm [18]. This requires introducing mechanisms for *conditional constraints*, which are created by the STPU compilation process. These are constraints that specify an edge weight that the dispatcher must enforce until an uncontrollable event executes, but the constraint is removed once the uncontrollable event executes.

5.1 Background on Simple Temporal Problems with Uncertainty

We begin with 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 occurs, the end event occurs sometime during the feasible duration, but is outside the control of the dispatcher. 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. To execute this uncontrollable duration correctly, the dispatcher must not restrict the times when the end event may execute beyond the restriction imposed by the $[30, 70]$ constraint.

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* [11].

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, tightening 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. □

The dispatch algorithms are largely similar to those presented in Section 4 because most of the burden imposed by the uncontrollability is undertaken by the compiler. 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 Compiling Plans with Uncertainty

Drake's compilation algorithm is, in essence, an update of Stedl's dynamic controllability algorithm to use labeled data structures [19]. Stedl's algorithm proceeds in essentially two steps. First, it computes the APSP form of the distance graph, ignoring the fact that some intervals are uncontrollable. Second, it performs additional reasoning, called back-propagation, that ensures that the types of tightenings discussed above cannot occur at run-time. The propagations in the second phase may create *conditional constraints*.

Definition 5.3 (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 $A - C \leq t$. □

Morris demonstrated that the addition of conditional constraints, which were first formulated as wait constraints, into the compiled form is sufficient to perform this reasoning and dispatch dynamically controllable STPUs. Wait constraints are the same as conditional constraints, except that the weight is negated. The compilation process for a STPU terminates with a distance graph that may include some conditional constraints. The key innovation of this section is the compact encoding defined below as *Conditional Labeled Distance Graphs with Uncertainty*. Drake maintains a separate labeled value set for each triple of events in order to store any conditional constraints, indicating the start, end, and conditional events of the constraint.

Definition 5.4 (Conditional Labeled Distance Graph with Uncertainty) A conditional labeled distance graph G is a tuple $\langle V, W, C\rangle$. V is a list of vertices representing the events. W is a set of labeled value sets for the weights and the marking of the controllability of the edge. The labeled value sets store pairs (a, b), where a is the weight and b is either C for controllable or U for uncontrollable, with domination function $f((a, b), (a', b')) \leftarrow (a < a')$. This set 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 contains those pairs of events (i, j) where $w(i, j) \neq \infty$, 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. □

Adapting the compilation algorithm itself follows the same procedure as earlier sections, namely, apply operations with the labeled function and handle any detected inconsistencies by creating conflicts. We use the following example to illustrate the compilation algorithm.

Example 5.5 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 initial problem, is first compiled into the APSP form shown in Figure 5.1a and then the back-propagation and filtering produce the result in Figure 5.1c. □

The APSP computation is unchanged from the previous algorithm. Reviewing the complete set of back-propagation rules, introduced by Stedl, in detail is beyond the scope of this paper, but it is helpful to explain the derivation of a conditional constraint in this example [19].

Example 5.6 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 instead as $\langle 60, B \rangle$ instead [13]. Stedl changed the notation into conditional constraints to make it consistent with the semantics of the distance graph, which we adopt here. □

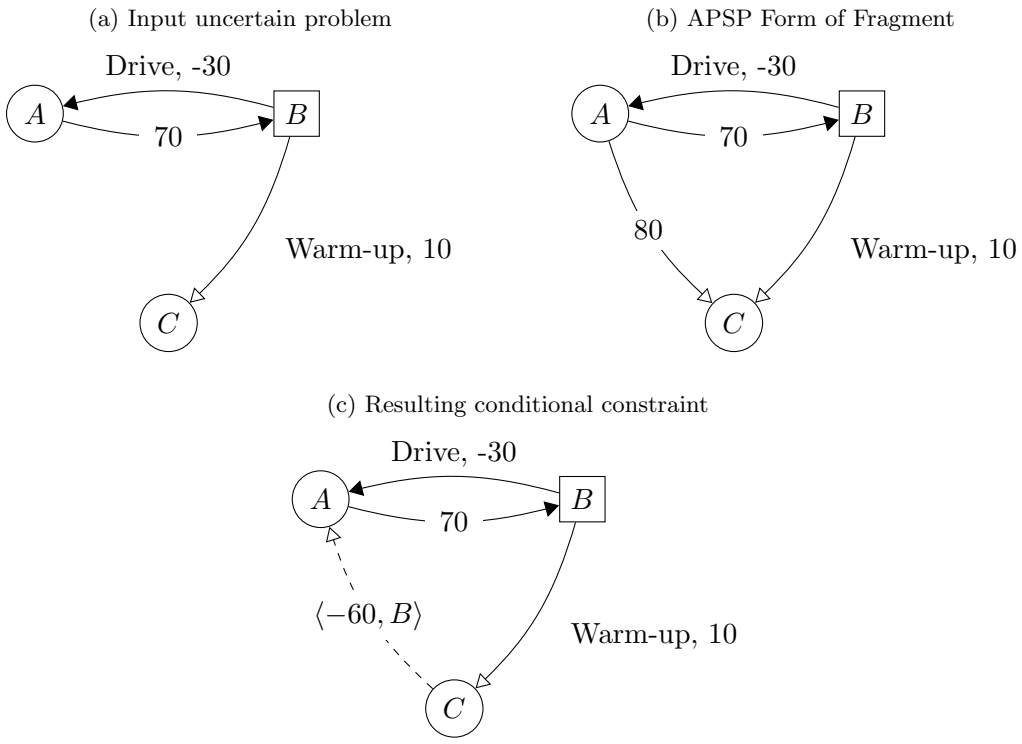
All possible such derivations are formalized in a set of back-propagation rules that Stedl's algorithm applies recursively to update the distance graph with the consequences of the uncontrollable durations. Using these back-propagation rules on labeled values simply involves placing the union of the 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 3.3, which performs this environment operation.

This completes our description of the compilation algorithm for problems with temporal uncertainty. We have explained a strategy for 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 details on our updated version of Stedl's controllability algorithm are provided in Appendix 5.2. As we have seen, given the techniques and data structures developed for the controllable case, extending Drake's compilation process is a relatively simple process.

5.3 Dispatching Plans with Uncertainty

Morris et al. proved that dynamically dispatching a compiled STPU requires two simple modifications to the STP dispatcher that Drake mimics [11]. 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 add another type of constraint that might be violated, which EventExecutable? must

Figure 5.1: An example of an uncertain duration and a conditional constraint that results. Depicts Examples 5.5 and Example 5.6.



be modified to find and collect the environments for, determining whether conflicts can be created without invalidating all possible choices. Otherwise, these extra constraints do not change dispatching. 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 Section 4, except that they must call the two modified functions. At this stage, these modifications pose little technical difficulty and do not provide any new insight. Therefore, we delay a detailed discussion until Appendix C.3.

5.4 Conclusions

This section completes the presentation of Drake's technical innovations. Previous sections have developed labeled value sets as a simple and efficient technique for applying non-disjunctive temporal reasoning to disjunctive problems. This section 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.

6 Experimental Results

This section 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 [24]. 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, with respect to direct enumeration. 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 at compile and dispatch.

We generated a suite of random DTPs, DTPUs, TPNs, and TPNUs, with generators documented in Appendix B. Our test suite includes DTPs and DTPUs with up to thousands of component STPs. Specifically, we generated 100 consistent problems at each parameter 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 value. We generated TPNs and TPNUs recursively, creating 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. For all types of problems, inconsistent instances were discarded.

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. Although organized differently, the operations required to insert values and perform operations are 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 it 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.

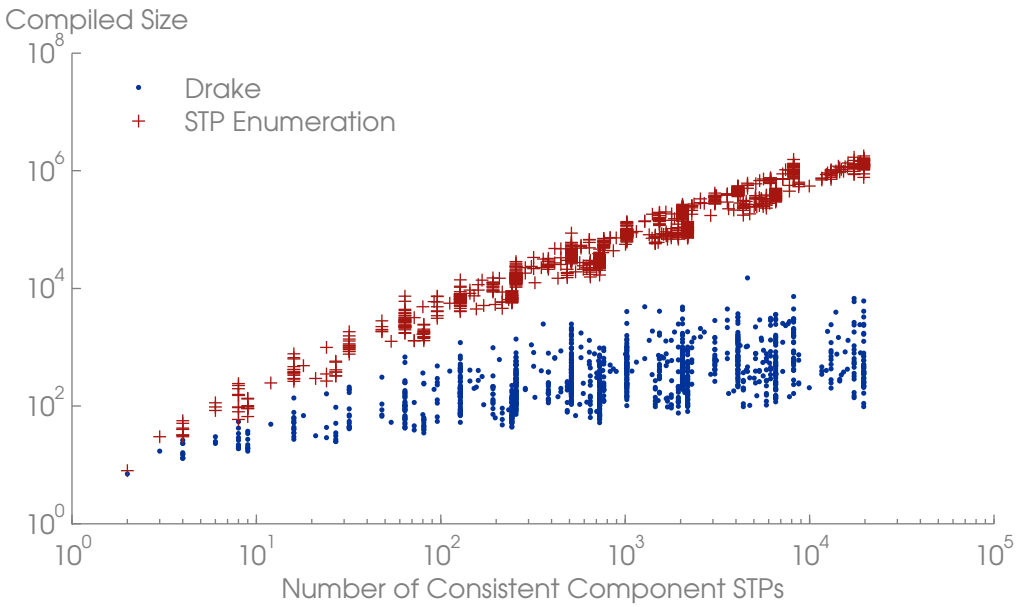
We begin with the size of the compiled representation, because it provides the clearest and most favorable results. The size is computed as a platform independent metric, counting all the important data structure elements stored by each representation. Drake's size metric counts the number of events, 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 6.1 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 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 the same as the DTP data. 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 primary factor. 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. Additionally, the additional cost of storing STPUs is simply the space needed 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 6.2, 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. However, there is a small, yet noticeable fraction

Figure 6.1: 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.

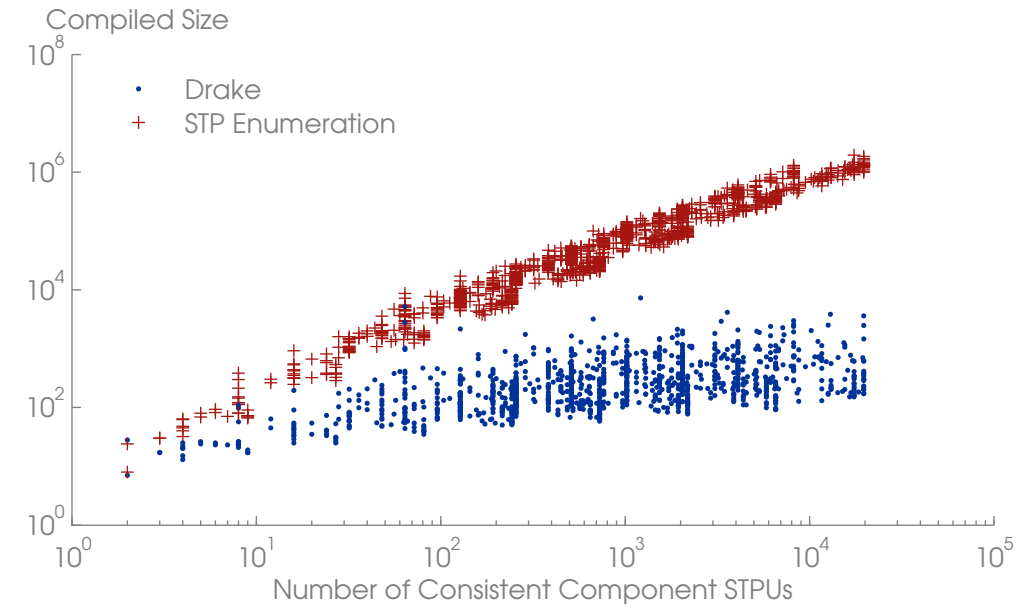
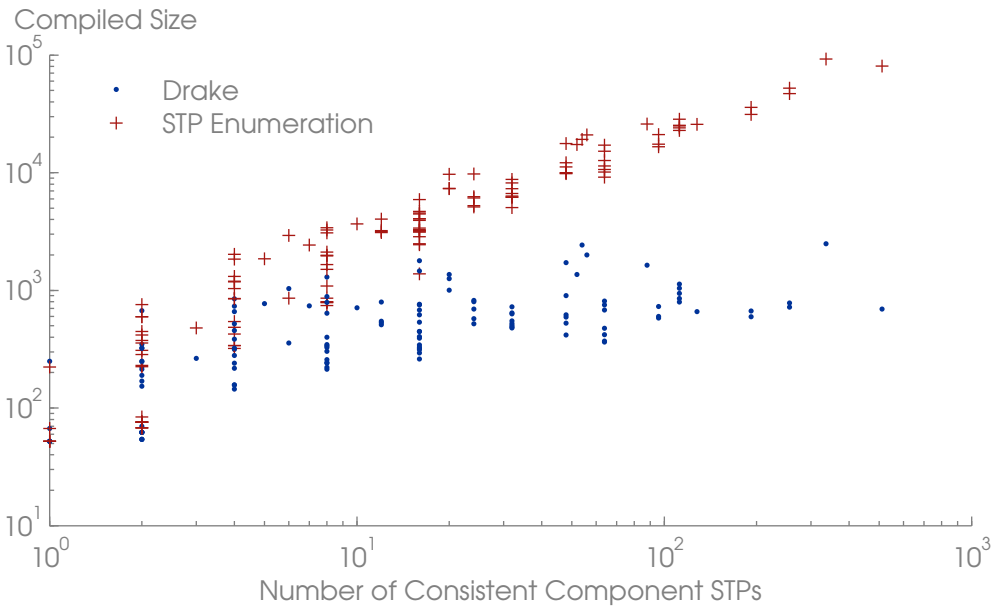


Figure 6.1: The compiled size of random problems as a function of the number of component STP(U)s (cont).

(c) The compiled size of TPNs.



(d) The compiled size of TPNUs.

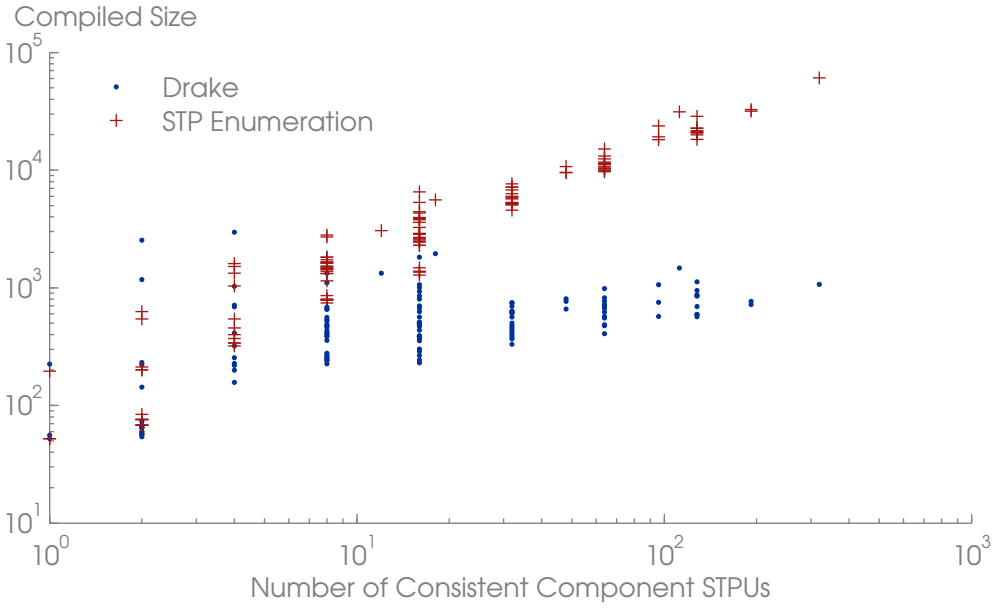
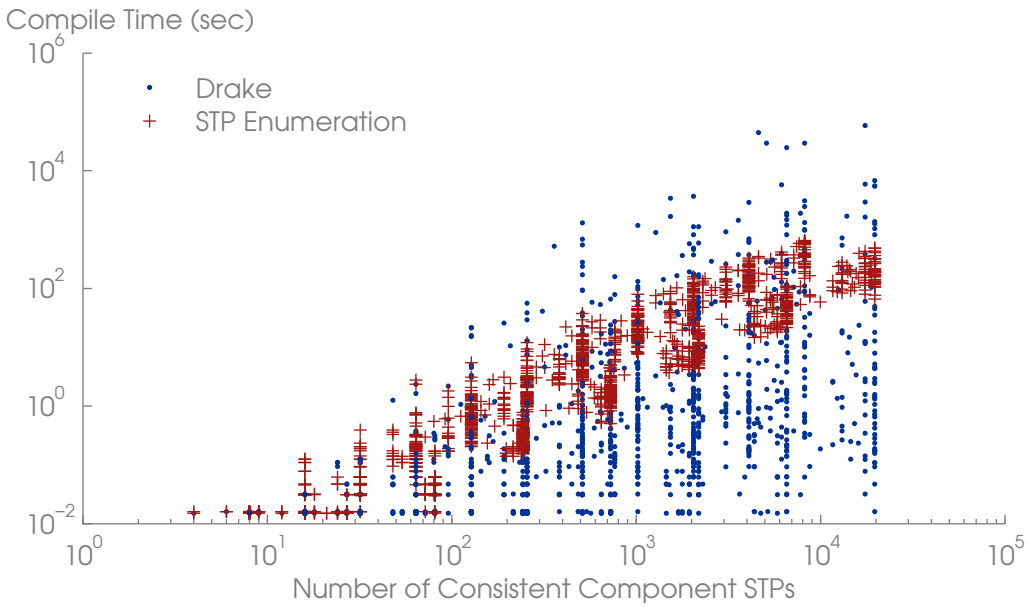


Figure 6.2: The compile time of random problems as a function of the number of component STP(U)s.

(a) The compile time of DTPs.



(b) The compile time of DTPUs.

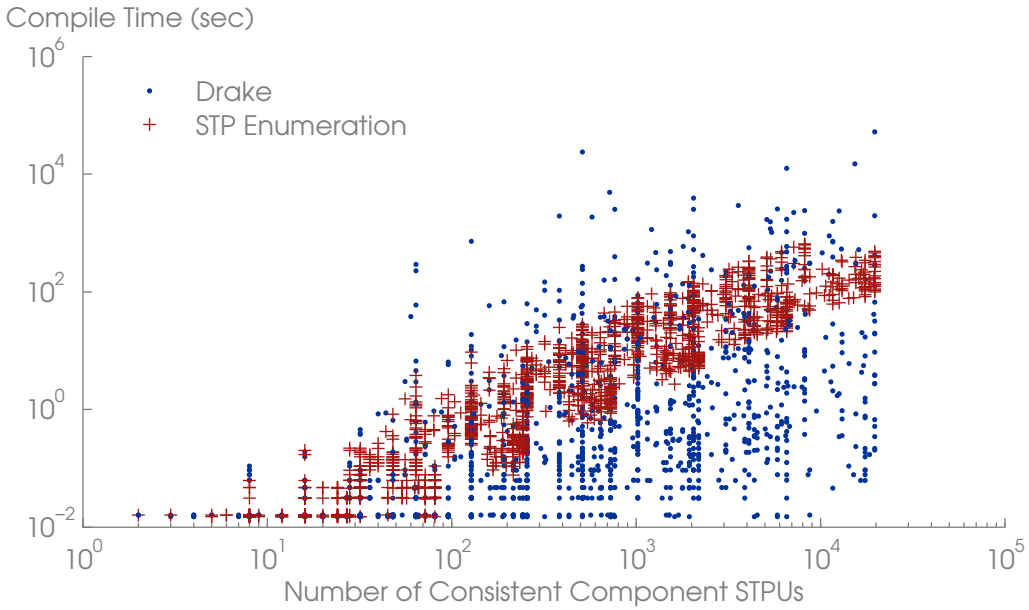
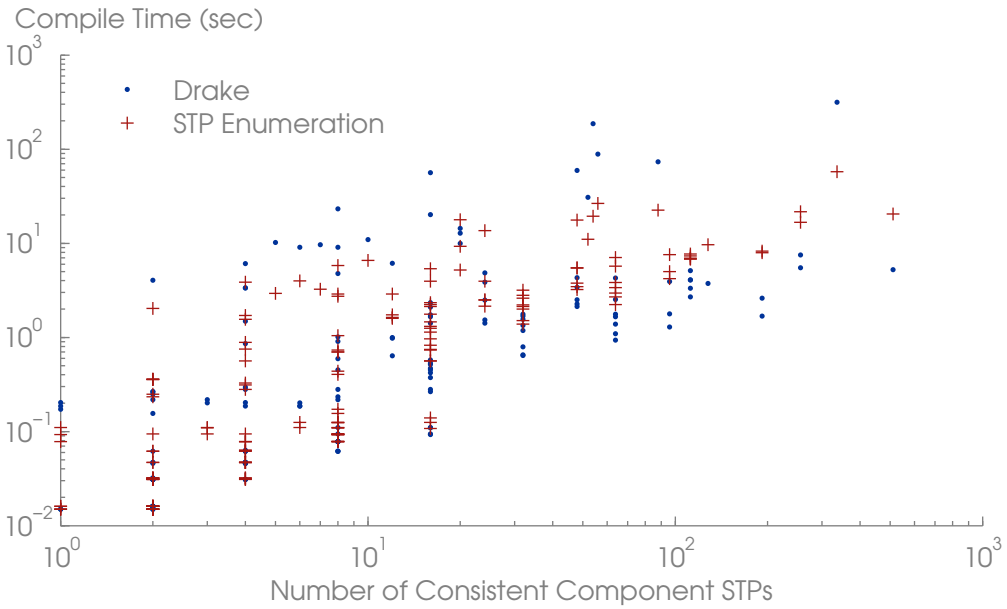
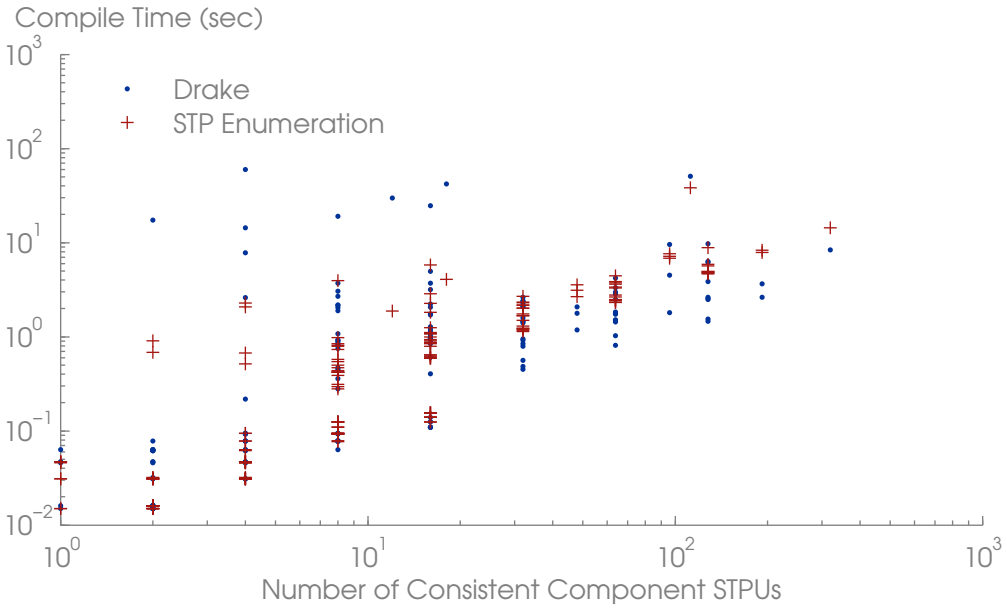


Figure 6.2: The compile time of random problems as a function of the number of component STP(U)s (cont).

(c) The compile time of TPNs.



(d) The compile time of TPNUs.



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.

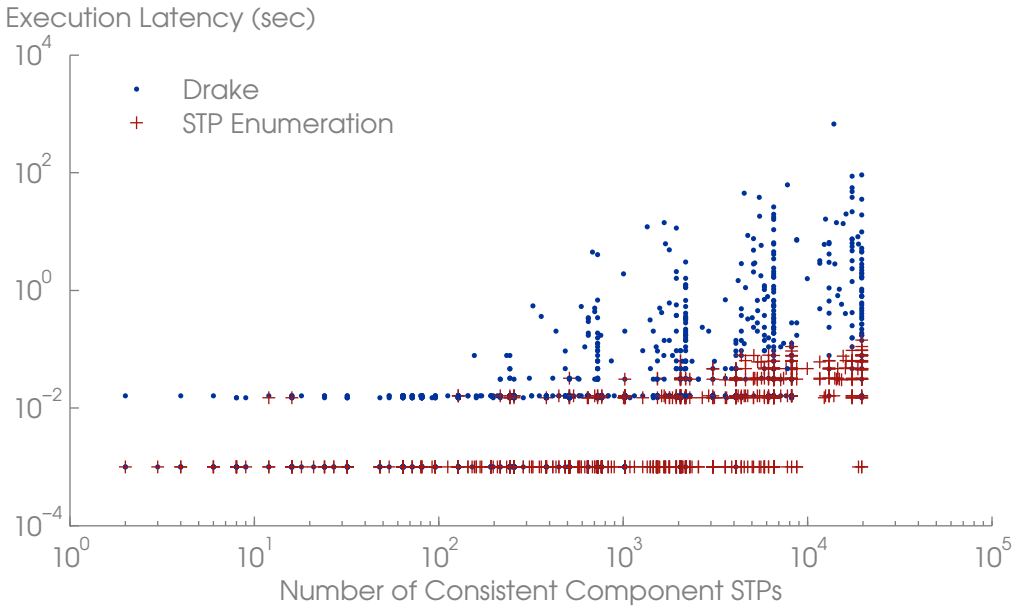
The final metric we present is execution latency. To collect this data, Drake simulated running the plans once, of which we recorded the longest decision making period, during which Drake selected events to execute and performed propagations. The STP enumeration strategy was timed for the first dispatch 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 6.3. 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 non-zero time.

Generally, Drake takes significantly more time to make decisions for large problems than Tsamardinos's approach, which we believe 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. 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 our 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.

Figure 6.3: 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.

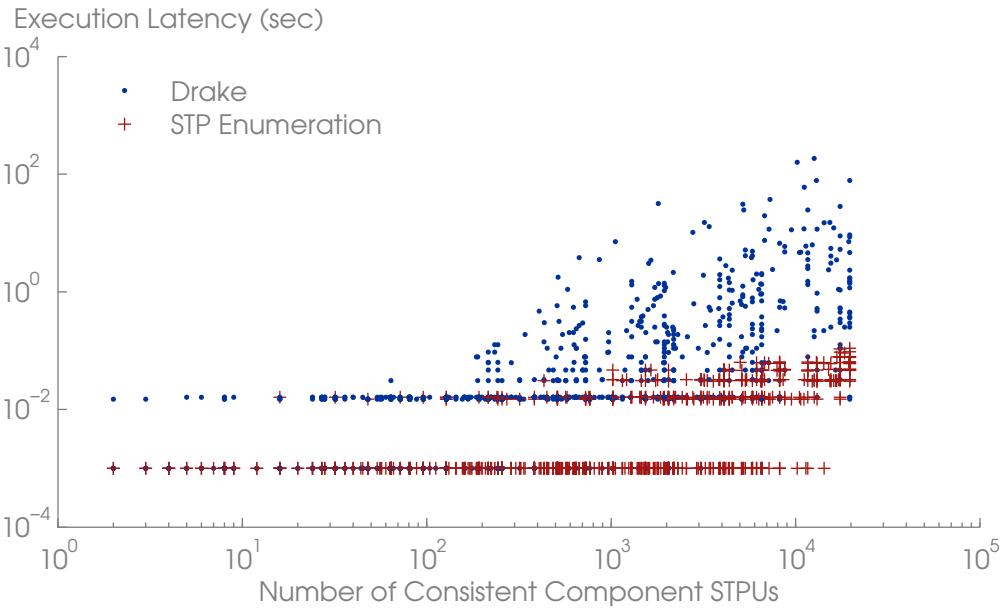
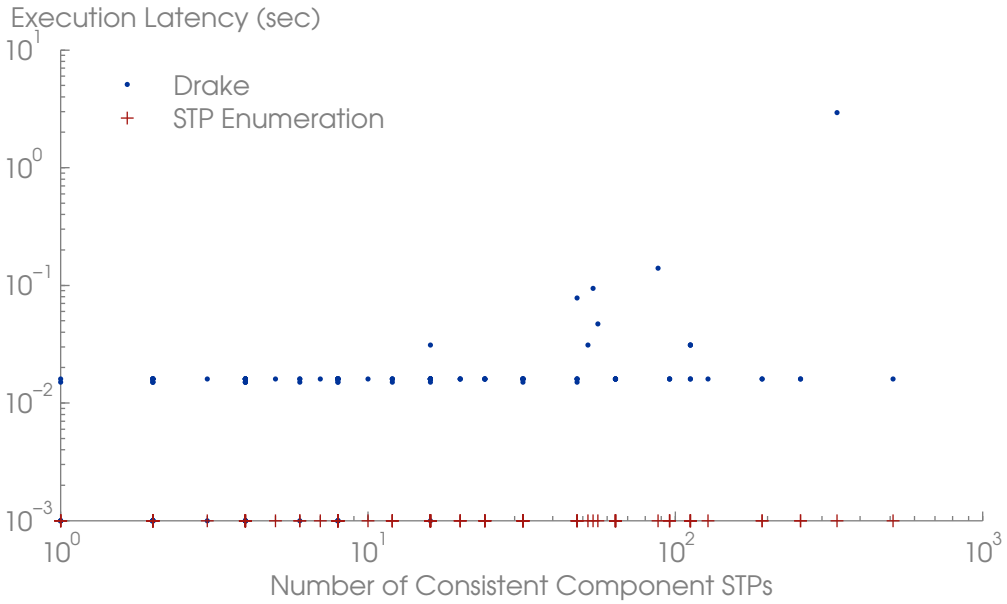
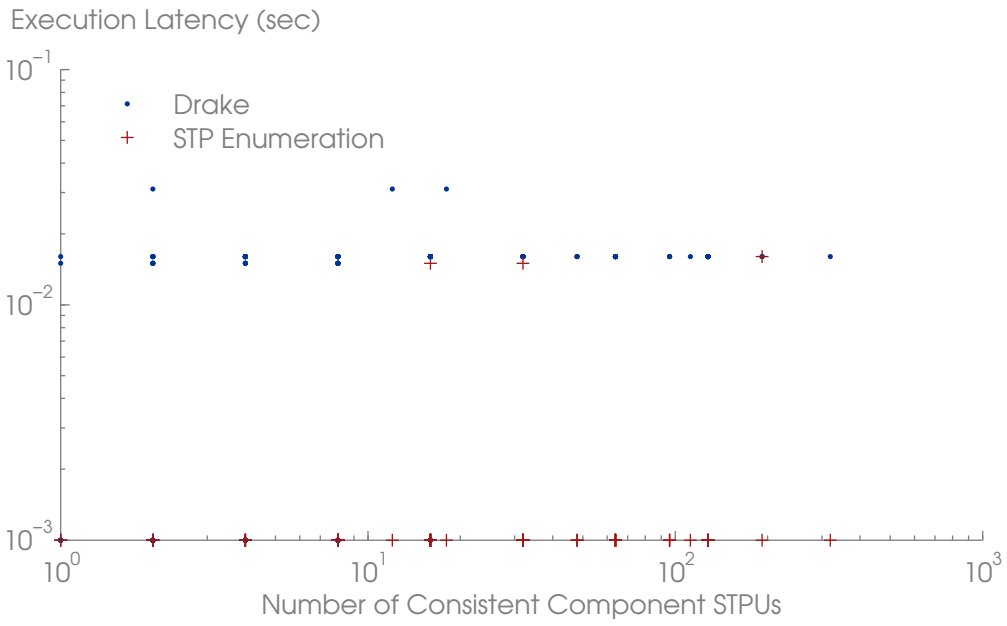


Figure 6.3: The execution latency of random problems as a function of the number of component STP(U)s (cont).

(c) The execution latency of TPNs.



(d) The execution latency of TPNUs.



7 Conclusions

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 decide the execution times and make discrete decisions at run-time [4, 9]. 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 [3]. 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 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 a new dynamic executive for TPNs and DTPs. 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.

8 Acknowledgements

The authors would like to thank Julie Shah for many helpful ideas and discussions. Patrick Conrad was funded during this work by a Department of Defense NDSEG Fellowship.

A Forming Labeled Distance Graphs

Building upon the 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 A.1 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. Section 3.3 defines environments formally, but an environment specifies a partial set of choices from which the constraint logically follows. Setting the environments in the initial encoding allows the algorithms to carry the choices through the later steps. 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 A.1 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 TPNToDGraph on it.

Example A.1 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 1.2. 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 1.1 can be recovered from this compact form, yet there are no duplicated constraints. □

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

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

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.

B Generating Random Problems

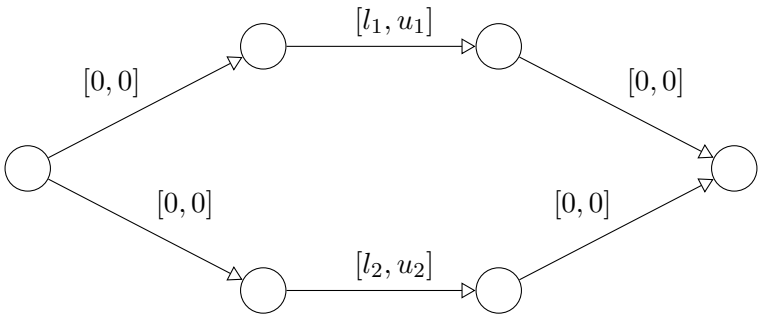
B.1 Generating Random DTPs

To generate random structured Disjunctive Temporal Problems, we modify Stedl's random structured STP generator [18]. 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 clauses per disjunctive constraint.

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



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 disjunctive constraints, so the increased number of disjunctive constraints do not become cluttered. 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 constraints. 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.

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.

B.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 B.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 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.

C Supplement to Plans with Uncertainty

C.1 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.

A TPNU is a TPN where some of the activities are marked as uncontrollable [1]. To perform constraint reasoning, we transform the TPNUs and DTPUs into labeled distance graphs with uncertainty. We now define Disjunctive Temporal Problem with Uncertainty, following Venable and Smith [25]. The deterministic DTP definition is augmented with uncontrollable events and edges. Prior literature often uses alternate terminology, referring to durations as requirement or contingent links.

Definition C.1 (Disjunctive Temporal Problem with Uncertainty [25]) 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.

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 Appendix A. 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. 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 Section 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.

C.2 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 runtime [11]. 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 [18]. 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

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

infeasible component problems. Therefore, we give an overview of the strategy and some algorithms here, but other details are omitted for brevity.

Stedl's top level fast dynamic controllability algorithm for STPUs works to compile families of STPUs, with only modifications for storing the edge weights in labeled value sets. The pseudo-code is shown in Algorithm C.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 C.1 Compilation algorithm for Labeled Distance Graphs with Uncertainty

```
1:  procedure CompileUncertain(V_c, V_u, W, S, C)
2:      W, S ← Labeled-Floyd-Warshall(V, W)
3:      if (¬EnvironmentsRemain?(S)) ∨ ¬PseudoControllable?(V, W, S) then ▷ Alg. C.2
4:          return null
5:      end if
6:      W ← FilterSTN(V, W)                                          ▷ See Section 3.6

7:      for v ∈ V_u do                                               ▷ propagate uncontrollable
8:          if ¬BackPropagateInit(V, W, S, C, v) then                ▷ See [18].
9:              return null
10:         end if
11:     end for

12:     if ¬PseudoControllable?(V, W, S) then                        ▷ Alg. C.2
13:         return null
14:     end if
15:     W ← FilterSTN(V, W)                                          ▷ See Section 3.6

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

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 component STPU is inconsistent, as before, because re-introducing the uncertainty makes the problem strictly harder and is certainly still inconsistent. Then Line 3 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 5.5 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 will never tighten 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 the literature [18]. 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. At this stage, uncontrollable durations may be pruned, as any uncontrollable durations the dispatcher needs are stored in the activity data structure.

Since pseudo-controllability is an important part of STPU compilation, we provide Algorithm C.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.

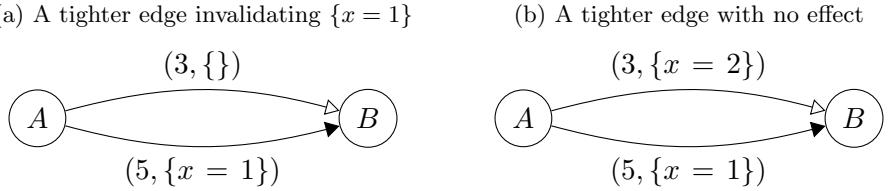
Example C.2 Figure C.1 shows two small graph segments we use to illustrate the pseudo-controllability algorithm. First, Figure C.1a has exactly one uncontrollable edge. The algorithm would look for any violating edges, immediately finding the only other edge from event A to B, which has a smaller weight. 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 C.1b 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 $\{x = 2\} \cup \{x = 1\} \neq \emptyset$. Therefore, the algorithm draws no new conclusions from this fragment. □

Algorithm C.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
```

Figure C.1: Graph fragments demonstrating pseudo-controllability.



C.3 Dispatching Algorithms for Uncontrollable Plans

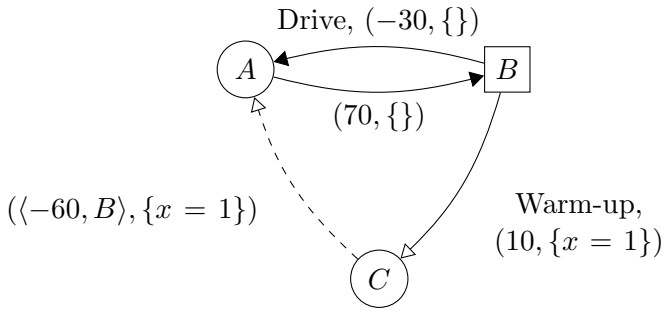
The most important update to the dispatcher is to modify the event selection routine to respect conditional constraints, meaning that to execute an event with a conditional constraint, either the conditional event has executed or the difference constraint is satisfied. In the labeled version, as before, 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 C.3 handles this extra consideration. Note that it is otherwise identical to Algorithm 4.3. Lines 20-28 loop over all the triples of conditional constraints that might restrict this event. Note that conditional constraints are always negative and point out from the constrained event. 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

Algorithm C.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
```

the conditional constraint is added to those that must be discarded and the algorithm proceeds as before.

Figure C.2: A fragment of a labeled conditional distance graph with uncertainty.



Example C.3 Consider executing the labeled conditional distance graph depicted in Figure C.2. 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 uncontrollable durations. Therefore, it also skips the computation of duration of the activity. Otherwise, the function is unchanged, identifying activities that should begin and computing execution times for controllable activities. 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 adding extra 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 C.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
```

D Proofs

PROOF (THEOREM 3.16) 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. ■

PROOF (THEOREM 3.17) 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. ■

PROOF (THEOREM 3.18) 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 3.17, 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. ■

PROOF (THEOREM 3.19) 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 correctly computed, as shown by Theorem 3.17. All the invalid component STPs are identified and discarded by negative self-loop edge weights, as in the unlabeled case. ■

PROOF (THEOREM 3.20) 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 the sum to subsume the environment for the value of (A, C), so that the sum holds in at least all the component STPs where the dominated edge holds. 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. ■

PROOF (THEOREM 4.5) Theorem 3.19 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 3.18, 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 3.16. 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. ■

PROOF (THEOREM 4.8) 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 3.19 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. ■

PROOF (THEOREM 4.15) 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. ■

References

[1] S.A. Block, A.F. Wehowsky, and B.C. Williams. Robust execution on contingent, temporally flexible plans. In *Proceedings of the 21st National Conference on Artificial Intelligence*, pages 802–808, 2006.

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

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

[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] L. Khatib, P. Morris, R. Morris, and F. Rossi. Temporal constraint reasoning with preferences. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, volume 1, pages 322–327, 2001.

[9] 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.

[10] D. McDermott. Contexts and data dependencies: A synthesis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, pages 237–246, 1983.

[11] 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.

[12] N. Muscettola. Computing the envelope for stepwise-constant resource allocations. In *Principles and Practice of Constraint Programming-CP 2002*, pages 109–119. Springer, 2006.

[13] 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.

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

[15] L. Planken, M. de Weerdt, R. van der Krogt, J. Rintanen, B. Nebel, J.C. Beck, and E. Hansen. P 3 C: A New Algorithm for the Simple Temporal Problem. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 256–263. AAAI Press, 2008.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] I. Tsamardinos. A probabilistic approach to robust execution of temporal plans with uncertainty. *Methods and Applications of Artificial Intelligence*, pages 751–751, 2002.

[22] 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.

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

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

[25] 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.

[26] 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.

[27] 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.