A Factored Planner for the Temporal Coordination of Autonomous Systems

by

David Cheng-Ping Wang

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2015 [June 2015]

Author ..... Signature redacted
Department of Aeronautics and Astronautics
May 21, 2015

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

Certified by ..... Signature redacted
Leslie P. Kaelbling
Professor
Thesis Supervisor

Certified by ..... Signature redacted
Saman P. Amarasinghe
Professor
Thesis Supervisor

Accepted by ..... Signature redacted
Paulo C. Lozano
Chair, Graduate Program Committee

A Factored Planner for the Temporal Coordination of Autonomous Systems

by

David Cheng-Ping Wang

Submitted to the Department of Aeronautics and Astronautics
on May 21, 2015, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Embedded devices are being composed into ever more complex networked systems, including Earth observing systems and transportation networks. The complexity of these systems require automated coordination, but planning for and controlling these systems pose unique challenges. Devices can exhibit stateful, timed, and periodic behavior. A washing machine transitions automatically between its wash cycles while locking its door accordingly. The interaction between devices can cause indirect effects and require concurrency. A UAV with a simple GPS-based auto pilot may refuse to take off until it has a GPS fix, and may further require that fix be maintained while flying its route. While many planners exist that support some of these features, to our knowledge, no planner can support them all, and none can handle automatic timed transitions.

In this thesis, we present tBurton, a domain-independent temporal planner for complex networked systems. tBurton can generate a plan that meets deadlines and maintains durative goals. Furthermore, the plan it generates is temporally least-commitment, affording some flexibility during plan execution.

tBurton uses a divide and conquer approach: dividing the problem into a directed acyclic graph of factors via causal-graph decomposition and conquering each factor with heuristic forward search. Planning is guided by the DAG structure of the causal graph, and consists of a recursive element. All of the sub-goals for a particular factor are gathered before generating its plan and regressing its sub-goals to parent factors. Key to this approach is a process we call unification, which exploits the locality of information afforded by factoring to efficiently prune unachievable sub-goal orderings before the computationally expensive task of planning.

The contributions of this thesis are three fold: First, we introduce a planner for

networked devices that supports a set of features never before found in one planner. Second, we introduce a new approach to factored planning based on timeline-based regression and heuristic forward search. Third, we demonstrate the effectiveness of this approach on both existing planning benchmarks, as well as a set of newly developed benchmarks that model networked devices.

Thesis Supervisor: Brian C. Williams
Title: Professor

Thesis Supervisor: Leslie P. Kaelbling
Title: Professor

Thesis Supervisor: Saman P. Amarasinghe
Title: Professor

Acknowledgments

I would like to thank Seung Cheung for his work in extending the original reactive Burton planner with qualitative time. His thesis laid the groundwork for this thesis.

Thank you to Professor Brian Williams, for his incredible patience and advice over the many years it has taken to commit this body of work to writing.

Thank you to Erez Karpas for his invaluable insights and encyclopedic knowledge of planning. His friendly and quick feedback has kept me from being mired in details or self-doubt and helped my writing move forward.

Thank you Howard Shrobe for his fatherly and reassuring guidance.

Thank you to my girlfriend of 12+ years, Justine Wang, whose support, patience, and enduring love knows no bounds.

Of course, thank you to the my committee members, Professor Leslie Kaelbling, and Professor Saman Amarasinghe for the meetings, understanding, and latitude.

This work was made possible through support from Lincoln Labs and the DARPA Mission-oriented Resilient Clouds (MRC) program.

Contents

Chapter 1

Introduction

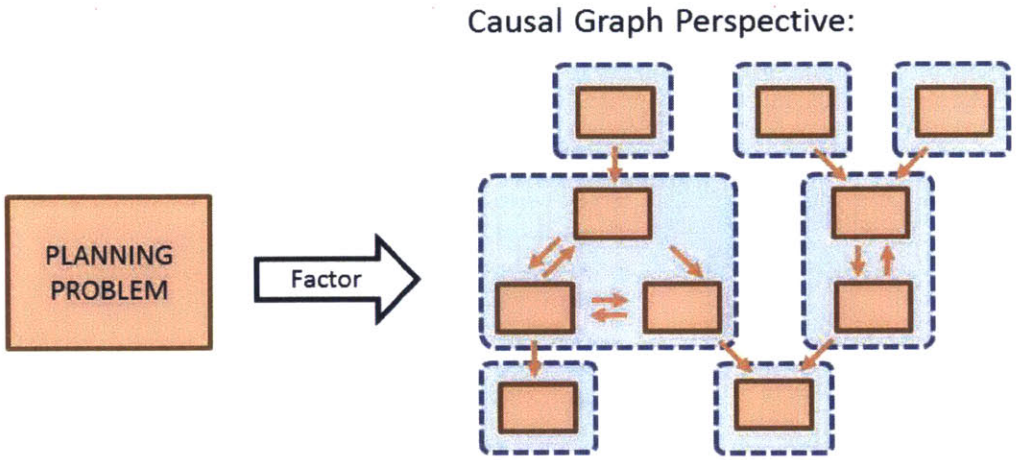
Embedded machines are being composed into ever more complex networked systems, including Earth observing systems and transportation networks. The complexity of these systems require automated coordination, but controlling these systems pose unique challenges: timed transitions – after turning a projector off a cool-down period must be obeyed before the projector will turn on again; periodic transitions – a satellite's orbit provides predictable time-windows over when it will be able to observe a phenomenon; required concurrency [14], – a communication channel must be open for the duration of a transmission. Furthermore, a user may wish to specify when different states need to be achieved (time-evolved goals) and may expect a plan that allows flexibility in execution time (a temporally least-commitment plan).

While there has been a long history of planners developed for these systems, no single planner supports this complete set of features. Model-based planners, such as Burton [57, 12], the namesake of our planner, have exploited the causal structure of the problem in order to be fast and generative, but lack the ability to reason over time. Timeline-based planners, such as EUROPA [22] and ASPEN [11], can express rich notions of metric time and resources, but have traditionally depended

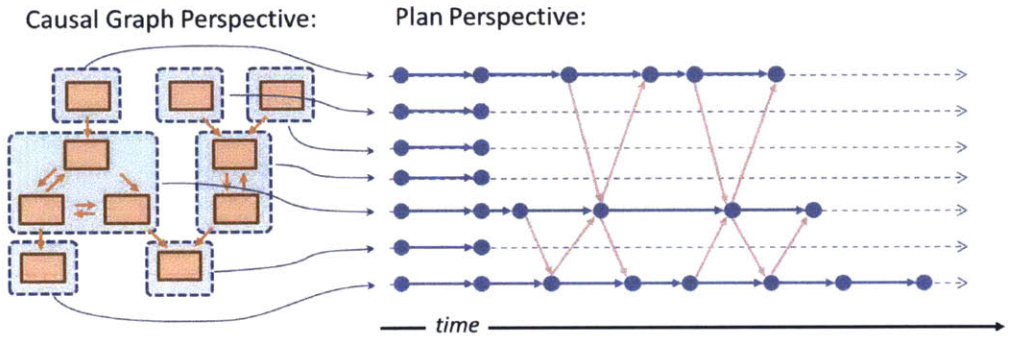
on domain-specific heuristics to efficiently guide backtracking search. Finally, metric-time heuristic planners [13, 4, 51] have been developed that are domain-independent, fast and scalable, but lack support for important problem features, such as timed and periodic transitions.

tBurton is a fast and efficient partial-order, temporal planner designed for networked devices. Our overall approach is divide and conquer, a.k.a factored planning [3], but we leverage insights from model, timeline, and heuristic-based planning. Like Burton, tBurton factors the planning problem into an acyclic causal-graph and uses this structure to impose a search ordering from child to parent (Figure 1-1(a)). Associated with each factor is a timeline on which a portion of the plan corresponding to that factor will be constructed (Figure 1-1(b)). Factors help maintain the locality of causal information, thereby reducing the number of time-consuming threat-detection steps common in partial-order planning. Timelines reflect that locality of information in the plan. By accumulating and ordering the goals for a factor along a timeline before attempting to achieve them, we can detect inconsistencies early. To find a plan, we use a conflict directed search method that leverages the efficiency of a heuristic-based sub-planner to completely populate the timeline of a factor, before regressing the sub-goals of that plan to the timeline of its parent (Figure 1-2).

The contributions of this thesis are three fold: First, we introduce a planner for networked devices that supports a set of features never before found in one planner. These features enable our planner to combine activity planning with the concurrent control of timed, discrete devices. Second, we introduce a new approach to factored planning based on timeline-based regression and heuristic forward search. This approach exploits structure to decompose, while solving hard subproblems by exposing inconsistencies early along timelines and guiding planning along these timelines. Third, we demonstrate the effectiveness of this approach on existing planning benchmarks, and introduce new benchmarks that are representative of problems that

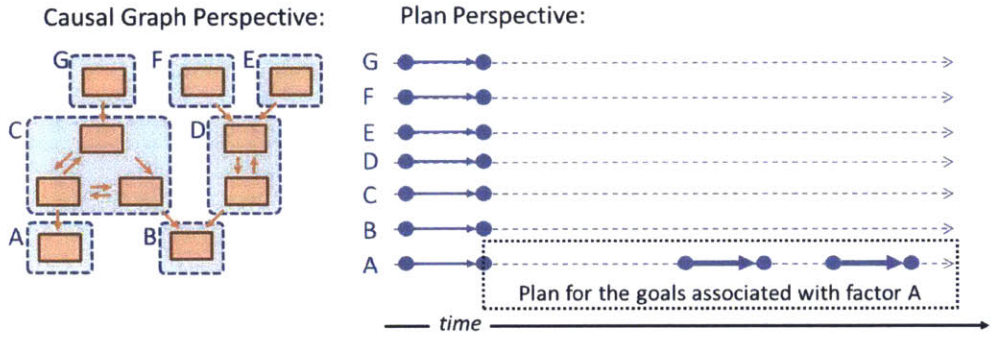


(a) The planning problem is factored into a causal graph. Automata are its basic building blocks and are depicted with orange rectangles. The orange arrows show causal dependencies between the automata. Cyclic dependencies are clustered into factors, shown in blue, to make the causal graph acyclic.

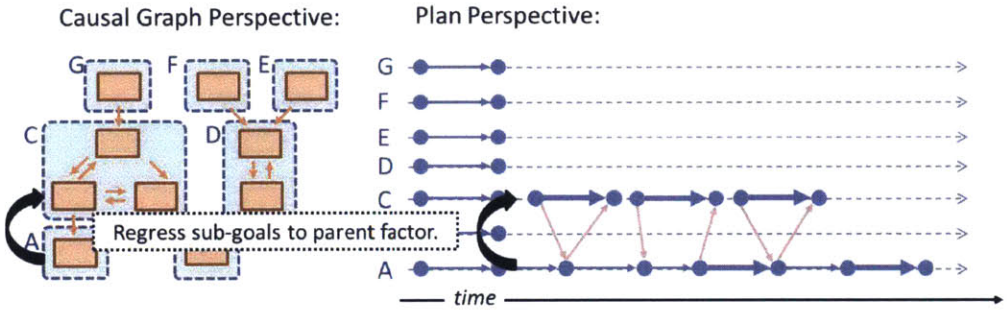


(b) tBurton reasons over two perspectives of the planning problem. The *causal graph perspective* provides the order in which sub-goals are regressed, but the plan is better represented along timelines. In the *plan perspective*, there is one timeline per factor. The blue arrows depict the plan for each factor. The red arrows indicate constraints between the progression of the plans of different factors.

Figure 1-1



(a) tBurton plans starting from the bottom of the causal graph. Since there are goals associated with factor A, planning starts with factor A.



(b) Once a plan for factor A is complete, the sub-goals required for that plan are regressed to its parent in the causal graph, factor C.

Figure 1-2: A simple two-step representation of tBurton's approach to planning.

combine concurrency, action, and device control.

We start this chapter with a pair of motivating examples, highlighting the problem features tBurton is capable of handling as well as its ancestry. We then provide an overview of our approach and sketch how we intend to handle those problem features. We situate this approach in the context of prior work. Finally, we highlight our experimental results.

1.1 Planning for the Real World

The past 20 years has seen a remarkable progression in the speed with which planners can generate a plan. From GraphPlan [6], to BlackBox [35], to the use of heuristics [32, 29, 49] and BDD [34] encodings, planners have gotten faster and faster. But, many of these techniques are developed for STRIPS [19] and ADL [24], foundational languages in the field of planning which cannot represent important features of real world problems. The STanford Research Institute Problem Solver (STRIPS) language was developed in 1971, ~45 years ago. The Action Description Language (ADL) was developed in 1987, ~30years ago.

The modern planning language of choice, which inherits from both STRIPS and ADL, is the Planning Domain Description Language (PDDL) [26]. Despite the extension of PDDL to include more real-world features, such as time [21], time-evolved goals, and preferences [25], planners that support these newer language features have been slow to evolve. More to the point, Boddy has argued the addition of time while maintaining classical STRIPS semantics, has limited the types of real world problems PDDL can represent [7].

In this section we describe two real-world scenarios that could greatly benefit from automated planning. We enumerate the important problem features they require that have yet to be commonly supported, and use these examples as motivations for a list

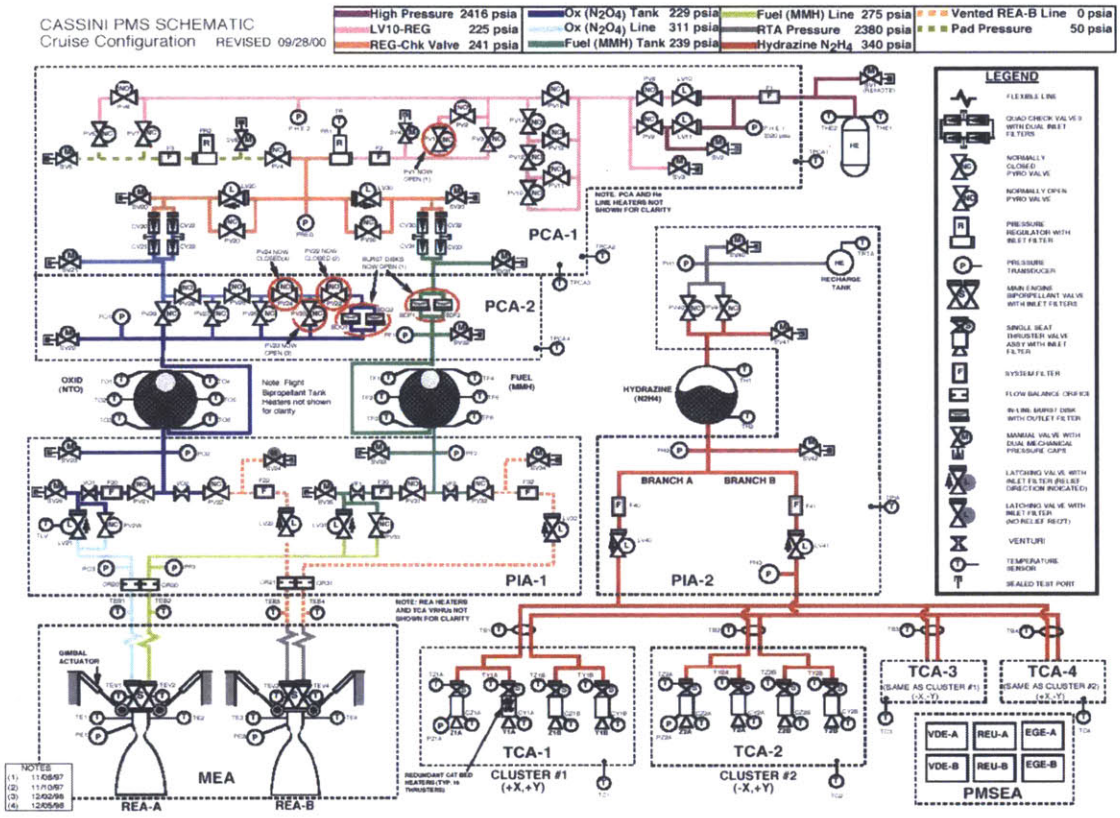


Figure 1-3: A diagram showing the complexity of the Cassini dual main-engine and attitude control thrusters.

of features that tBurton supports.

One of the examples involves controlling the Cassini-Huygens probe, perhaps the most complex interplanetary probe humanity has ever launched. The other involves a more human touch, where firefighters and their equipment must interact to put-out wild fires. We use this latter firefighting scenario as an example of the inputs and outputs to the planning problem tBurton solves.

1.1.1 Controlling Cassini

The Current Generation

In 1997, after $\sim$ 20 years of development, the Cassini-Huygens probe was launched to explore Saturn and its moons [41]. It successfully inserted itself into Saturn orbit in 2004, deployed its lander in 2005, and continues to operate well-beyond its 4 year primary mission to study Saturn and its moon Titan.

At the time of its launch, the European Space Agency touted Cassini-Huygens as was one of the largest, heaviest, and most complex interplanetary spacecraft ever built. Figure 1-3 shows the Cassini Propulsion Module Subsystem (PMS), a network of tubing controlled by over 50 valves backed by electronic drivers, to redundantly bleed, pressurize, and fire Cassini's dual main engines and attitude control thrusters.

Controlling such a complex spacecraft is a very difficult task, and has historically depended on active human involvement. Unfortunately, since it takes 1-3 hours for round-trip communication time, active human involvement takes the form of a lot of reasoning on the ground, representing that thought in a script, and uploading it. Cassini's nominal behavior is scripted, with checkpoints requiring ground-based approval. Its fault recovery depends on man-in-the-loop replanning and scripted recovery behaviors. The success of which, both depend on the imagination, experience, and diligence of humans.

As a prime example, one month after Cassini launched, it was discovered that the Prime Regulator (LV10-REG), a critical and non-redundant component in the PMS had failed to completely close, resulting in a leak. Ground teams had to design a new script for Cassini's Saturn orbital insertion burn, and upload it mid-flight. The script was designed to only open certain valves right before ignition to prevent the unnecessary loss of the helium pressurant.

The Next Generation

As the complexity of these spacecraft increase, the current practice of man-in-the-loop reasoning and scripted behaviors has become untenable. While humans are very capable problem solvers, we have difficulty remembering all of the interactions of complex systems. Reasoning over such large interacting systems demands some degree of autonomy.

Planning has the potential to not only generate the mission scripts, but provide dynamic, real-time recovery from a diversity of faults. Burton, the predecessor of tBurton, demonstrated this concept on NASA's Deep Space One autonomous spacecraft [57].

Burton's design philosophy was to remove search from on-line execution, thus allowing the algorithm to reconfigure the spacecraft as quickly as possible. While successful, this philosophy limited Burton's problem representation.

Burton represented the world using a set of interacting state-machines called Concurrent Constraint Automata, but required that all actions be reversible (infinite 'undos'). It further required that it be fed one goal-state at a time, to help ensure the next activity is returned within a constant response time. Subsequent work to expand Burton resulted in a planner that could achieve more than one goal at a time, and reason over a CCA and plan with qualitative time ('before', 'after'), but showed that the use of metric time would require search [12].

To solve real-world problems, we need richer representations. Take, for example the Cassini PMS. We need to reason over irreversible actions: Many of the valves are pyrotechnic, and cannot be closed once opened and vice-versa. We need to reason over time: Actions like pre-heating the main-engines take time, and so does scheduling orbital insertion. We need to reason over indirect effects (which Burton, through CCA, already supports): Opening a valve when other valves are not configured cor-

rectly can cause premature ignition. We need to be able to control concurrent systems concurrently: Valves may need to be transitioned simultaneously to reduce leakage.

tBurton supports these problem features, and leverages insights from Burton to speedup the planning process. In the next subsection we situate these features in an example related to firefighting. We them sketch out the problem tBurton solves.

1.1.2 Firefighting in the Wild

Planning is not only useful for interplanetary spacecraft, planning can also make an impact closer to home. Each year wild fires "impact watersheds, ecosystems, infrastructure, business, individuals, and the local and national economy" [15], incurring far-ranging costs beyond the millions spent in taming them. But, fighting a wild fire is not as simple as dumping water. Today's wild fire battles are strategically planned by experienced fire teams, and orchestrated with ground-teams, helicopters, and even firefighting planes such as the Bombardier CL-415. These situations require both the firefighters and their equipment to act in concert, blending considerations for both man and machine into the planning problem.

Figure 1-4 depicts an example firefighting scenario. There are 5 isolated fires threatening a population center. Fires C and D are already endangering the population. But, a shift in the wind is expected in a few hours that will drive the windward fires A and B towards the population center. 2 ground teams, a helicopter, and 2 Bombardier 410s are available to fight the fires, but need to be coordinated. Ground teams, denoted by firemen's helmets, need to communicate, but line of sight communication is blocked by a mountain range. The helicopter is capable of facilitating that communication but it can also douse fires with a foam retardant. Finally, the 2 planes can put out fires using water refilled from the two local lakes.

Just as the complexity of controlling spacecraft can benefit from planning, so can

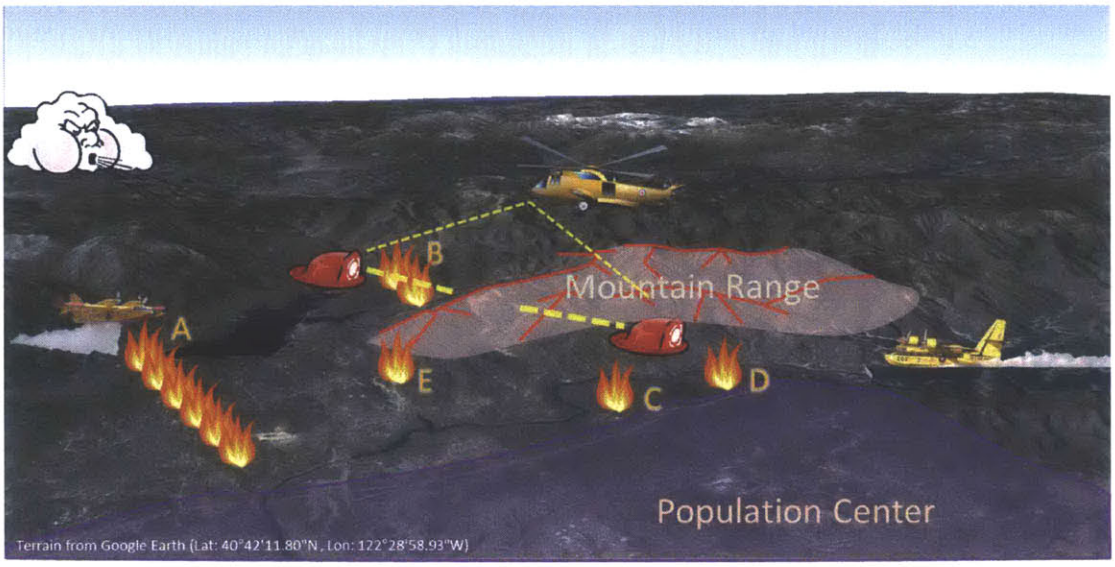


Figure 1-4: The firefighting scenario.

the coordination of firefighters.

1.1.3 Supported Problem Features

tBurton is capable of solving planning problems expressed with many features important to real-world problems. While controlling interplanetary spacecraft and fighting wild fires sound worlds apart, they share many of these features in common. We categorize these features in terms of whether they enhance the expressiveness of goal specification, behavior modeling, or the plan representation.

Goal Features

tBurton is capable of planning for multiple goals expressed over finite periods of time. These are sometimes called time-evolved or temporally-extended goals. In our list of supported goal features, we delineate between two different types of goals, as they are commonly delineated in operations research, even though tBurton represents both

types of goals similarly.

- **Goal Deadlines** express that a goal must be achieved by a given time. For example, fire C and D both need to be extinguished in 1 hour, before they reach the population center. Additionally, fire A needs to be put out before the wind starts.

- **Goal Maintenance** express that a goal must be achieved for at least a specified duration. For example, the helicopter needs to hold position over the mountains to act as a relay from time X until the firefighters are done extinguishing fire E.

Modeling Features

tBurton represents the world by using a timed extension to Burton's CCA models, akin to timed automata.

- **Actions and State Machines**. Human activities are most naturally modeled as separate actions, such as run, jump, and walk. But, it is often easier to describe devices in terms of state machines. tBurton can reason over both representations. [1]

- **Indirect Effects**. In STRIPS style planning, the planner is assumed to the sole agent of change, selecting which actions to incorporate into the plan. An indirect effect allows an action to be conditioned on a state (or another action). Thus, while the model is still deterministic, the use of an action in the plan could cause a chain reaction of actions that must be accounted for in the plan. For example, moving the helicopter could immediately terminate communications between the ground-teams for which it is acting as a relay.

[1] tBurton internally represents the world by using a variation on timed automata theory, but in Chapter 6 we describe how to map PDDL's action-style representation to our own.

- **Timed transitions** allow a modeler to express actions and transitions which take time. For example, its important to model the amount of time it takes to refill the water tank on the planes.

  As a variation on timed transitions, tBurton can also reason over **autonomous timed transitions**. These are actions and transitions that will occur automatically given enough time. For example, a fire that is not fully extinguished might automatically re-ignite after a period of time.

The combination of indirect effects and autonomous timed transitions allow the modeling of chain-reactions, where a single action causes a cascade of events.

Plan Features

The plan tBurton outputs maintains concurrency of activities while giving them flexible execution times.

- **Concurrency** in a plan is desirable for two reasons. First, it reduces the duration (make-span) of the plan by arranging actions in parallel. From an alternate perspective, it keeps available resources simultaneously and efficiently engaged, rather than using them in sequence. In our firefighting scenario, fighting the fire would be most efficient if both ground-teams, both planes, and the helicopter could act concurrently. Second, concurrency is sometimes required for the planning problem [14]. If one action can only take place during other actions, the plan must be able to represent their concurrent execution.

- **Temporally Flexible**. Once a plan is generated, it is usually executed. But, executing a plan is difficult in an uncertain world. A temporally flexible plan uses time-windows to express when actions/transitions should happen instead of time-points. This allows some flexibility in when those activities are executed.

Figure 1-5: tBurton's Inputs and Outputs

In our firefighting scenario, temporal flexibility is important if the fire takes longer to put out than expected, or if headwinds slow the rate at which the planes can shuttle water from the lake to the fire.

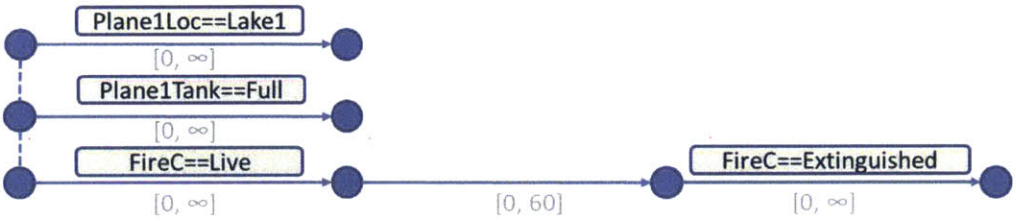
1.1.4 Example Problem Formulation

tBurton supports all of these features through a novel problem formulation and a unique approach. To our knowledge, no other planner is capable of singularly supporting all of these features. In the remainder of this section we formulate how these features are represented. In the next section we will describe our approach and sketch out the reasoning process for these features.

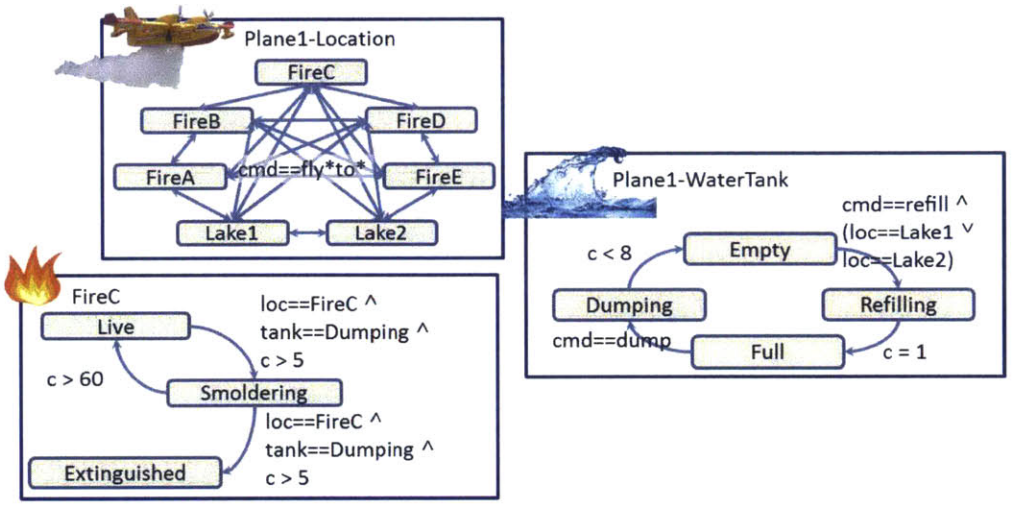
Like any planner, tBurton takes as input some specification of the initial state of the world, desired goals, and a model representing possible behaviors, to produce a plan. However, the manner in which tBurton represents these inputs and outputs goes a long way towards supporting these real-world features.

Figure 1-5 depicts tBurton's inputs and outputs. One input tBurton takes is an *initial state plan*, a representation that combines both the initial state and desired goals and lays them out across a timeline. The other input is a model, expressed as a set of interacting timed automata. tBurton outputs a *total state plan*, an elaboration of the initial state plan that lays-out the behaviors of the model needed to achieve the goals.

Take a simple subset of our firefighting scenario as an example. A plane needs to shuttle water from a nearby lake to douse a fire.



(a) Initial State Plan for Firefighting Scenario



(b) Automata-based Model for Firefighting Scenario

Figure 1-6: Example inputs to the tBurton planner.

Modeling

Figure 1-6(b) depicts the behaviors relevant to this problem using tBurton's native automata representation. Each of the three boxes in this figure depicts an important state of our problem as an automaton. The box labeled "FireC" depicts the behavior of fire C. In this simple model, the fire can either be 'Live', 'Smoldering', or 'Extinguished'. The box labeled "Plane1-Location" depicts where the plane can fly. The box labeled "Plane1-WaterTank" depicts its various discrete capacities. Transitions, denoted by blue arrows, indicate the various behaviors of our system. For the fire to

transition from 'Live' to 'Smoldering', a plane over that location must dump water on the fire for at least 5 minutes. This colloquial statement is expressed formally by labeling the transition with a guard, a statement that must be true to affect the transition. The passage of time is expressed using the clock variable 'c', which ticks along with real-world time. Note the transition from 'Smoldering' to 'Live' is guarded only by time. If the fire is 'Smoldering' and is not doused again in 60 minutes, it will flare up and become 'Live' again.

This automata modeling formalism natively allows for the expression of indirect effects and timed transitions. Automata interact by guarding the transition in one automaton by the location of another automaton. For example, the transition from fire C 'Live' to 'Smoldering' requires the plane be at location 'FireC'. Suppose our goal is to dump the plane's water tank just prior to returning to base, if the plane happens to be at FireC, it could cause the indirect effect of dousing the fire.

Timed transitions are any transitions guarded by time, as indicated by a clock variable. The transitions related to dousing the fire all require 5 minutes, and thus are timed. But, perhaps most interesting, are the automated timed transitions, where the transition is only guarded by time. In our example: The fire can flare up after 60 minutes.

Another modeling feature tBurton supports is an action-based instead of automata-based representation. This alternate representation is supported by a mapping from the action-based PDDL representation to the automata representation we have just described. We explore this mapping in detail in Chapter 6.

Expressing Plans

tBurton expresses both its input initial state and goals as well as its output in the same formalism. While there are a few stipulations as to what is required for the

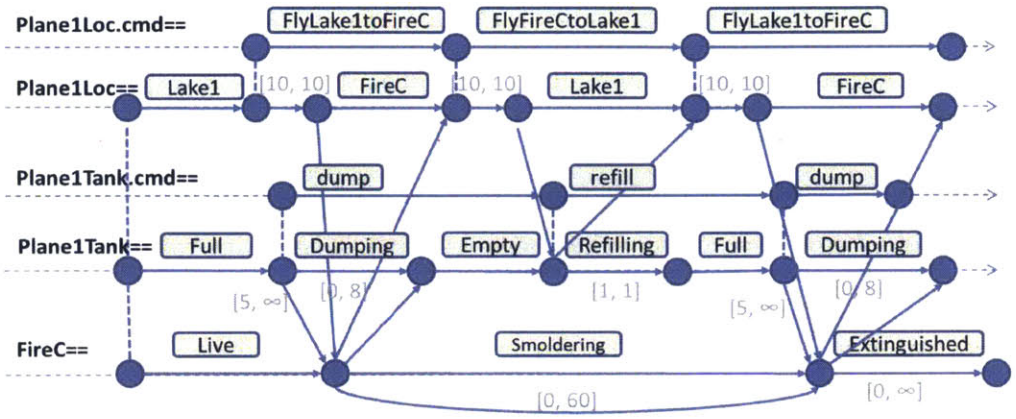


Figure 1-7: Example output from the tBurton planner: A plan for the Firefighting Scenario

initial state plan and output *total state plan*, the most important difference is that the total state plan must achieve all goals in the initial state plan.

Figure 1-6(a) depicts an initial state plan for the firefighting scenario, which requires FireC be extinguished in no later than 60 minutes. In this figure, the blue circles represent points in time and the blue arrows represent durations of time between those points. The durations are labeled with a time interval and can also be labeled with a statement. Together, the statement and time-interval describe what needs to be true and for how long. Alone, a duration with a time interval simply bounds the amount of time that can pass.

For an initial state plan, a duration must exist that specifies the initial state for each automaton, otherwise the remaining durations and statements specify the goals. Goal maintenance is easily expressed by labeling a duration with a statement of the goal. Goal deadlines, like extinguishing the fire in 60 minutes, can be expressed by preceding a goal maintenance statement with a bounded duration.

Figure 1-7 shows the total state plan that successfully extinguishes fire C. Each timeline depicts the evolution of a value important to our model. The timelines for 'Plan1Loc', 'Plane1Tank', and 'FireC' correspond to the automata. The timelines

for 'Plane1Loc.cmd' and 'Plane1Tank.cmd' correspond to special variables that our planner can set to control the transitions of the automata. The total state plan, while complex, shows the complete evolution of our system as it achieves its goals. In addition to the necessary actions of assigning the command variables, the total state plan also shows the expected system state.

From this plan, we can read that in order to get the Fire to transition from 'Live' to 'Smoldering' to 'Extinguished', the plane must fly from 'Lake1' to 'FireC' twice in order to douse the fire twice.

The various blue arrows temporally constrain the plan so these activities all occur at the correct times in relation to each other. These bounded temporal constraints afford the plan temporal flexibility. The structure of the plan, as laid out across multiple timelines, affords concurrency.

Now that we have an idea of how to express the problem, we return to the question of how we solve it.

1.2 Approach Overview

Divide and conquer is the basic principal behind factored planning, but alone tells only part of the story. As a factor planner, tBurton must decide how to factor (divide) the problem, how to plan for (conquer) each factor, *and* how to unify those plans.

Divide

Key to tBurton's approach to factored planning is exploiting the benefits of causal-graph based factoring in partial-order, temporal planning.

tBurton inherits its causal reasoning strategy from namesake, Burton [57, 12], a reactive, model-based planner developed for NASA spacecraft. Burton, exploits the near-DAG structure of its domain and grouped cyclically-dependent factors to

maintain an acyclic causal graph. The causal graph is then used to quickly determine a serialization of sub-goals. Even though this strategy is not optimal in the general case, complexity analysis has shown it is difficult to do better in the domain independent case [9, 8].

Despite the lack of optimality, the clustering of variables identified by an acyclic causal-graph has important ramifications. Goal-regression, partial-order planners [47, 61] traditionally suffer from computationally expensive threat detection and resolution, where interfering actions in a plan must be identified and ordered. Factoring inherently identifies the complicating shared variables, reducing the number of cases that must be checked for interference. Furthermore, threat resolution is equivalent to adding temporal constraints to order sub-plans during composition – reducing the number of threats under consideration also reduces the difficulty of temporal reasoning.

Conquer

In order to plan for each factor, tBurton uses a heuristic-based temporal planner. Heuristic-based planners, and especially heuristic forward search planners have scaled well [51, 13, 55], and consistently win top places in the IPC. Using a heuristic forward search planner (henceforth sub-planner) allows tBurton to not only benefit from the state-of-the-art in planning, but to degrade gracefully. Even if a problem has a fully connected causal-graph, and therefore admits only one factor, the planning time will be that of the sub-planner plus some of tBurton's processing overhead.

tBurton plans for each factor by first collecting and ordering all the goals for that factor along its timeline. The sub-planner is then used to populate the timeline by first planning from the initial state to the first goal, from that goal to the next, and so on. The problem presented to the sub-planner only contains variables relevant to

that factor.

Unify

Sub-plans are unified by regressing the subgoals required by a factor's plan, to parent factors. Early work in factored planning obviated the need for unification by planning bottom-up in the causal-graph. Plans were generated for each factor by recursively composing the plans of its children, treating them as macro-actions [3, 9]. This approach obviated the need for unifying plans at the cost of storing all plans. Subsequent work sought to reduce this memory overhead by using backtracking search through a hierarchical clustering of factors called a dtree [36]. While tBurton does not use a dtree, we do extend backtracking search with plan caching and conflict learning in order to more efficiently unify plans.

1.3 Chapter Overview

In the next chapter, Chapter 2, we formalize our problem statement. In Chapter 3, we describe the high-level search which divides, conquers, and unifies. In Chapter 4 we describe the unify step in detail through the examination of the problem of Timeline Unification and its solution. In Chapter 5 we describe a component algorithm of unify called Incremental Temporal Consistency, which efficiency checks the consistency of the temporal constraints that form the basis of our plan. Chapter 6 describes the mapping from PDDL to TCA, which allows our planner to solve problems in the action representation used by PDDL. Chapter 7 presents our experimental results. Finally, Chapter 8 sketches out future work and concludes.

Chapter 2

Problem Statement: Timeline Planning for Concurrent Timed Automata

In this chapter, we formalize the problem formulation we sketched out in Chapter 1. We start by describing the 'project example', which we will use to clarify our definitions.

2.1 Projector Example

A running example we will use in the remainder of this thesis involves a computer, projector, and connection between them, which are needed to give a presentation. The computer exhibits boot-up and shutdown times. The projector exhibits similar warm-up and cool-down periods, but will also shutdown automatically when disconnected from the computer. Figure 2-1 depicts this example in tBurton's native, automata formulation.

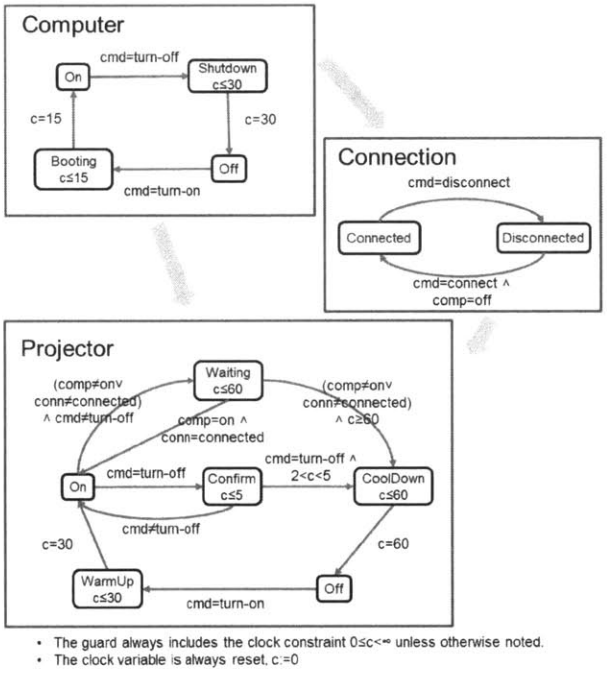


Figure 2-1: A simple computer-projector system represented as Timed Concurrent Automata.

2.2 Problem Formulation

Timed concurrent automata, the representation tBurton uses for the planning problem inherits from prior work on extending Concurrent Constraint Automata (CCA) [58] to time [33]. Our representation can be viewed as a variation on timed-automata theory [2], where transitions are guarded by expressions in propositional state-logic rather than symbols from an alphabet.

Formally, the planning problem tBurton solves is the tuple, $\langle TCA, SP_{part} \rangle$, where TCA is a model of the system expressed as a set of interacting automata called *Timed Concurrent Automata*, and SP_{part} is our goal and initial state representation, which captures the timing and duration of desired states as a *Partial State Plan.* We represent our plan as a Total State Plan SP_{total}, which is an elaboration of SP_{part} that contains no open goals, and expresses not only the control commands needed, but the resulting state evolution of the system under control.

2.2.1 Timed Concurrent Automata

Definition 1. a TCA, is a tuple $\langle L, \mathcal{C}, \mathcal{U}, \mathcal{A} \rangle$, where:

- $\mathcal{L}$ is a set of variables, $l \in \mathcal{L}$, with finite domain $D(l)$, representing locations within the automata. An assignment to a variable is the pair (l, v), $v \in D(l)$.
- $\mathcal{C}$ is a set of positive, real-valued clock variables. Each clock variable, $c \in \mathcal{C}$, represents a resettable counter that increments in real-time, at the same rate as all other clock variables. We allow the comparison of clock variables to real valued constants, $c \; op \; r$, where $op \in \{\leq, <, =, >, \geq\}$, $r \in \mathbb{R}$, and assignments of real-valued constants to clock variables $c := r$, but do not allow clock arithmetic.
- $\mathcal{U}$ is a set of control variables, $u \in \mathcal{U}$, with finite domain $D(u) \cup \perp$. $D(u)$ represents *commands* users 'outside' the TCA can use to control the TCA. $\perp$

is a nullary command, indicating no command is given.

- $\mathcal{A}$ is a set of timed-automata, $A \in \mathcal{A}$.

A TCA can be thought of as a system where the location and clock variables maintain internal state while the control variables provide external inputs. The automata that compose the TCA are the devices of our system.

Definition 2. A single timed automaton A is the 5-tuple $\langle l, c, u, \mathbb{T}, \mathbb{I} \rangle$.

- $l \in \mathcal{L}$ is a location variable, whose values represent the locations over which this automaton transitions.
- $c \in \mathcal{C}$ is the unique clock variable for this automaton.
- $u \in \mathcal{U}$ is the unique control variable for this automaton.
- $\mathcal{T}$ is a set of *transitions* of the form, $T = \langle l_s, l_e, g, c := 0 \rangle$, that associates with a start and end location $l_s, l_e \in D(l)$, a guard g and a reset value for clock c, $c := 0$. The guard is expressed in terms of propositional formulas with equality, φ, where: $\varphi ::= \texttt{true} \mid \texttt{false} \mid (l^o = v) \mid (u = v) \mid (c \; op \; r) \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$. The guard can be expressed only in terms of location variables not belonging to this automaton, $l^o \in \mathcal{L} \backslash l$, and the control and clock variable of this automaton. For brevity, we will sometimes use the expression $l \neq v$ in place of $\neg(l = v)$. The automaton is said to instantaneously transition from l_s to l_e and reset its clock variable when the guard is first evaluated true.
- $\mathbb{I}$ is a function that associates an invariant with each location. The invariant takes the form of a clock comparison $c < r$ or $c \leq r$ that bounds the maximum amount of time an automata can stay in that location. The invariant $c \leq \infty$ expresses that an automaton can dwell in the associated location indefinitely.

In the projector example (Figure 2-1), the projector has locations `Off`, `WarmUp`, `On`, `Waiting`, `Confirm`, and `CoolDown`. The transitions, represented by directed edges between the locations, are labeled by guards that are a function of other location variables, clock variables (denoted by `c`), and control variables (denoted by `cmd`). Invariants label some states, such as `WarmUp`, which allows us to encode the bounded-time requirement that the projector must transition from `WarmUp` to `On` in 30.

An automaton is *well formed* if there exists a unique next-state for all possible combinations of assignments to location, control, and clock variables. With regards to the transitions, an automaton is said to be *deterministic* if for any l_s only the guard g of one transition can be true at any time. Figure 2-2 depicts the set of next-states from the `Waiting` state of the well-formed, deterministic Projector automaton. Note that there is exactly one possible next-state for each combination of inputs, and a unique next-state at the upper-bound of the invariant. In order to produce plans with predictable behavior, tBurton must plan over TCA models consisting of well-formed, deterministic automata.

TCAs make representing problem features such as indirect effects, automatic-timed transitions, and periodic transitions straight forward. An *indirect effect* can occur when a transition in automaton A_c is *activated by* (the guard of the transition is evaluated true based on the assignments to) the locations of other automaton $\mathcal{A}_p$ without the need to assign the control variable of A_c. *Automatic-timed transitions* can occur within one automaton when a transition is activated by the clock variable. *Periodic transitions* can occur when a series of automated-timed transitions starts and ends with the same location (forming a cycle). A combination of these features can produce more complex interactions: several indirect effects could produce a chain-reaction, and indirect effects with automatic-timed transitions can result in periodic transitions spanning several automaton.

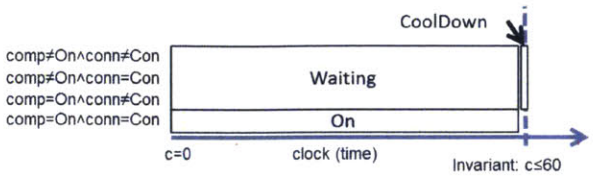


Figure 2-2: A graphical representation of the possible set of next-states from the `Waiting` state of the well-formed and deterministic Projector automaton.

2.2.2 State Plans

To control TCA, we need to specify and reason about executions (runs) of the system in terms of state trajectories; we represent these as state plans. A state plan specifies both the desired (goal) and required (plan) evolution of the location and command variables using a set of timelines we refer to as state *histories*. The temporal aspect of these histories and the relationship between events within them are expressed through simple temporal constraints [16]. The flexibility afforded by simple temporal constraints allows the state plan to be a temporally least commitment specification, which is important in enabling plan executives [39, 43] to adapt to a range of disturbances without the need to re-plan. tBurton generalizes upon the value and justification episode and history representation employed in TCP [59], which introduced a least-commitment approach to representing interactions between timelines. The use differs in that TCP performs progression on histories through state constraints, as a form of least-commitment simulation, while tBurton performs regression on goal histories through timed concurrent automata, as a form of least-commitment planning.

Conceptually, a state-plan can be thought of as one large temporal network which we divide into histories and justifications. Each history describes the evolution of a particular location or control variable, while justifications are temporal constraints that relate the timing of one history to another.

Definition 3. a history H is a tuple $\langle EV, EP \rangle$ where:

$EV = \{e_1, e_2, ..., e_n\}$, is a set of events e_i (that represent positive, real-valued time points), and

$EP = \{ep_1, ep_2, ..., ep_m\}$. is a set of episodes, where each episode, $ep = \langle e_i, e_j, lb, ub, sc \rangle$, associates with a temporal constraint $lb \leq e_j - e_i \leq ub$, a state constraint sc expressed as a propositional formula over location and control variable assignments. We refer to e_i and e_j as the *start* and *end events/times* of the episode, respectively, even though their actual temporal ordering depends on whether lb is positive.

Definition 4. a state plan SP is a tuple $\langle GH, \mathcal{VH}, \mathcal{J} \rangle$ where:

- GH is a *goal history*, a type of history in which the state-constraints of episodes represent the *desired* evolution of location and control variables.
- $\mathcal{VH}$ is a set of value histories, $VH \in \mathcal{VH}$. Each *value history*, VH, is a type of history that represents the planned evolution of locations or control variables. The state constraints in a value history are restricted to contain only variable assignments and only assignments to the variables to which the value history is associated. As such, a value history represents the trace of the corresponding variables.
- $\mathcal{J}$ is a justification, a type of episode with a state constraint of value `true`, which is used to relate the temporal occurrence of events in goal histories to events in value histories.

For simplicity, we will use the term *goal episode* to refer to an episode occurring in the goal history, and the term *value episode* for an episode occurring in the value history.

As with many partial-order planners, tBurton searches over a space of partial plans, starting from the partial state-plan SP_{part} and elaborating the plan until a

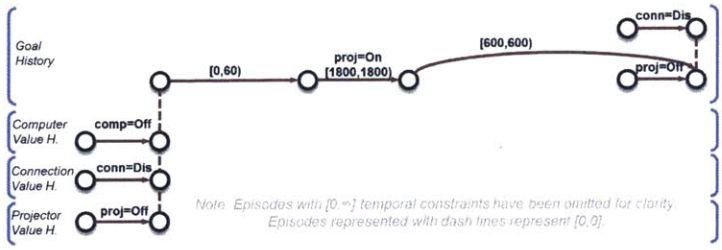
valid total plan SP_{total}, in which all goals are closed, is reached. State plans allow tBurton to not only keep track of the plan (through value histories), but also keep track of why the plan was created (goal histories and justifications). This is useful for post-planning validation, but also allows tBurton to keep track of open-goals during the planning process.

Figure 2-3 shows a progression between the initial state plan and the total state plan for the projector example. Figure 2-3(a) depicts the partial state plan from which tBurton starts planning. The presenter wishes to project for 30 minutes and leave the room 10 minutes later with projector off and his computer disconnected. The goal history of the state-plan, as indicated by the brackets, describes this desired state trajectory. The value histories represent the initial state of the computer, connection, and projector. Figure 2-3(a) shows the output of tBurton, a total state plan, where the histories track the evolution of each variable's value. The emphasized commands of 'connecting the computer and projector', 'turning on the projector', and then 'disconnecting the computer and projector' represent the actionable control-variable assignments needed to achieve the presenter's goals.

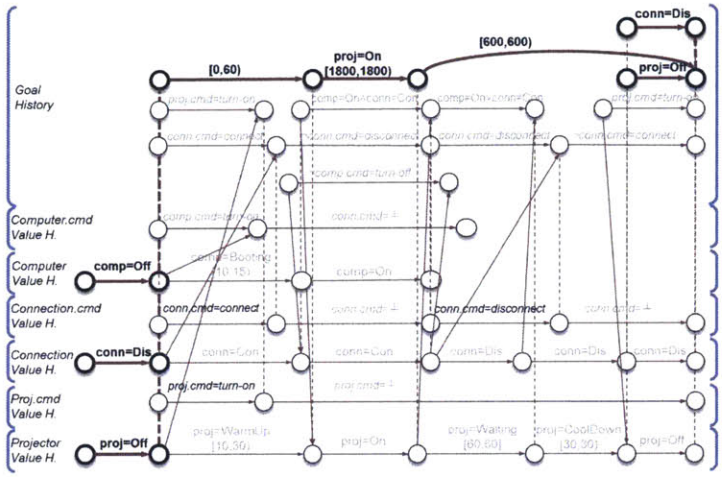
2.2.3 Semantics of TCAs and State Plans

Until now, we have used the words *run*, *trace*, *closed*, *valid*, *least-commitment*, and *causal graph* in a general sense. We now return to formalize these definitions, starting with the semantics of $TCAs$ and their relation to state plans.

Definition 5. run. The *run* of a single timed-automaton, A_a, can be described by its timed state trajectory, $S_a = ((l_{a0}, 0), (l_{a1}, t_{a1})..., (l_{am}, t_{am}))$, a sequence of pairs describing the location and time at which the automaton first entered each state. l_{a0} is the initial assignment to the location variable of A_a. We say a run $((l_{ai}, t_{ai}), (l_{aj}, t_{aj}))$ is *legal* if two conditions are met. First, if l_{ai} has invariant guard $c < r$, then $t_{aj} - t_{ai} < r$



(a) The initial partial state plan for the projector example. Events are represented as vertices. Episodes as edges. The lower and upper time bounds for episodes are labeled using interval notation.



(b) The total state plan for the projector example. The control variables important to the execution of the state plan as well as the initial state plan are darkened.

Figure 2-3: Initial and total state plan for the projector example.

must be true. Second, if there exists a transition between locations l_{ai} to l_{aj}, guarded by g, the first time g became true in the interval from t_{ai} to t_{aj} must be at time t_{aj}.

Definition 6. trace. A *trace* in timed automata theory is usually defined as the timed-trajectory of symbols, a word. Relative to TCAs, this corresponds to the timed trajectory of the guard variables. For $TCAs$, where automata asynchronously guard each other through their location variables, a trace and run are almost identical, with the exception that a trace for a TCA would also include the timed state trajectory of

control variables used in guards. A trace for a TCA therefore captures the evolution of all the variables.

Definition 7. closed. A goal-episode with constraint sc_g is considered trivially closed if sc_g evaluates to true, and trivially un-closable if it evaluates to false. Otherwise, a goal-episode is considered *closed* if there is a co-occurring episode in a value-history whose constraint entails the goal's constraint. Formally, a goal-episode $\langle e_{gs}, e_{ge}, lb_g, ub_g, sc_g \rangle$ is closed by a value-episode $\langle e_{vs}, e_{ve}, lb_v, ub_v, sc_v \rangle$, if $sc_v \models sc_g$, and the events are constrained such that $e_{vs} = e_{gs}$ and $e_{ve} = e_{ge}$. A goal appearing in the goal-history which is not closed is *open.*

In the context of a state-plan, $\langle GH, \mathcal{VH}, \mathcal{J} \rangle$: The goal-episode occurs in the the goal history GH. The value-episode occurs in a value history, $VH \in \mathcal{VH}$. And, we represent closed by adding two justifications to $\mathcal{J}$, which constrain the two episodes to start and end at the same time.

Definition 8. valid. In general, SP_{total} is a *valid* plan for the problem $\langle TCA, SP_{part} \rangle$, if SP_{total} has no *open* goal-episodes, *closes* all the goal-episodes in SP_{part}, and has value-histories that both contain the value history of SP_{part} and is a legal trace of the TCA.

For tBurton, we help ensure SP_{total} is a valid plan by introducing two additional requirements. First, we require that SP_{part} contains an episode in the value history of each location variable, l, whose end event must precede the start event of any goal on l, thus providing a complete 'initial state'. Second we require that SP_{part} be a subgraph of SP_{total}. These two additional requirements allow us to simplify the definition of valid: SP_{total} is a valid plan if it has no open goals and is a trace of the TCA. Figure 2-3(b), which depicts the total state plan for the projector example, uses bold lines to emphasize the subset of episodes that came from the partial plan.

To ensure SP_{total} is temporally, *least commitment*, the state plan must be consistent, complete, and minimal with respect to the planning problem. A valid state plan is already *consistent* and *complete* in that it expresses legal behavior for the model and closes all goals. We consider a state plan to be *minimal* if relaxing any of the episodes (decreasing the lower-bound, or increasing upper-bound) in the histories admit traces of the TCA that are also legal. Structurally, the definition of a state plan (def. 4) ensures tBurton cannot return plans containing disjunctive state constraints in the value histories, nor can the plan contain disjunctive temporal constraints.

Definition 9. causal graph. The causal graph of a TCA is a digraph $\langle \mathcal{A}, E \rangle$ consisting of a set of automata embedded as vertices and a set of causal edges. A causal edge (A, B) is a member of E iff $A \neq B$ and there exists a transition in automaton B guarded by the location variable, l of automaton A. We say A is the *parent* of B, and equivalently, B is the child of A.

Chapter 3

Factored Timeline Planning based on Causal Graphs

3.1 tBurton Planner

tBurton's fundamental approach is to plan in a factored space by performing regression over histories; this is supported through several algorithms. Practically, tBurton's search involves: 1. Deciding which factor to plan for first. This scopes the remaining decisions by selecting the value history we must populate and the goal-episodes we need to close. 2. Choosing how to order the goal-episodes that factor can close 3. Choosing a set of value-episodes that should be used to close those goal-episodes. 4. Choosing the plans that should be used to achieve the value-episodes. 5. And finally, extracting the subgoals of the value history (guards) required to make the value-history a legal trace of the automata and adding corresponding goal-episodes to the goal history. These steps repeat until a plan is found, or no choices are available.

As a partial order planner, tBurton searches over variations of the state plan. Since we use the causal graph to define a search order, and subgoal extraction requires no

search, tBurton only has three different choices with which to modify SP_{part}.:

1. **Choose a goal ordering.** Since actions are not reversible and reachability checking is hard, the order in which goals are achieved matters. tBurton must impose a total ordering on the goals involving the location of a single automaton. Recall that since an automaton can have no concurrent transitions, a total order does not restrict the space of possible plans for any automaton. Relative to SP_{part}, imposing a total order involves adding episodes to the goal history of the form $ep = \langle e_i, e_j, 0, \infty, true \rangle$, for events e_i and e_j that must be ordered.

2. **Choose a value to close a goal.** Since goals can have constraints expressed as propositional state-logic, it is possible we may need to achieve disjunctive subgoals. In this case, tBurton must select a value that entails the goal.

   To properly close the goal, tBurton must also represent this value selection as an episode added to the value history of the appropriate automata or control variable, and introduce justifications to ensure the new episode starts and ends with goal episode (as described in definition 7).

3. **Choose a sub-plan to achieve a value.** The sub-plan tBurton must select need only consider the transitions in a single automaton, A. Therefore, the sub-plan must be selected based on two sequential episodes, ep_s ep_g, in the value history of A (which will be the initial state and goal for the sub-plan), and the set-bounded temporal constraint that separates them. tBurton can use any black box approach to select this sub-plan, but we will use a heuristic forward search, temporal planner. To add this sub-plan to SP_{part}, tBurton must represent the sub-plan in the value history and introduce any new subgoals this sub-plan requires of parent automata as additional goal episodes.

Adding a sub-plan to a value history is a straightforward process of representing

each state in the plan as a sequential chain of episodes $ep_1, ep_2, ..., ep_m$ reaching from the end event of ep_s to the start event of ep_g. Introducing subgoals is a bit trickier. Subgoals need to be computed from all value episodes added to represent the subplan, ep_i, as well as ep_g (for which we did not introduce goals as a result of the previous type of plan-space action, choosing a value to close a goal). The purpose of these subgoals is to ensure the parent automata and control variables have traces consistent with the trace (value-history) of A. There are two types of subgoals we can introduce: One is a maintenance related subgoal, that ensures A is not forced by any variables it depends on to transition early out of ep_i. The other expresses the subgoal required to effect the transition into ep_i.

To formalize the modifications to SP_{part}, let $ep1 = \langle e_{s1}, e_{e1}, lb_1, ub_1, sc_1 \rangle$ be an episode ep_s, ep_i, or ep_g in the value history of the location variable l of A, for which we need to introduce goals. Let $ep2$ be the episode that immediately follows $ep1$. Since $ep1$ (and similarly for $ep2$) is in the value history, we know from definition 4 that sc_1 and sc_2 are assignments to l. From the TCA model, lets also identify the set of transitions of the form, $T = \langle l_s, l_e, g, c := 0 \rangle$ [def. 2], for which sc_1 is the assignment $l = l_s$.

Adding a maintenance subgoal for ep_2 requires the following additions: A new goal episode for each T where sc_2 is not the assignment $l = l_e$, of the form $ep_{new} = \langle e_{s1}, e_{e1}, 0, \infty, l \neq l_e \rangle$. Adding a subgoal to effect the transition from ep_1 to ep_2 requires both a new goal episode and a justification. We add one goal episode for T where sc_2 is the assignment $l = l_e$, of the form $ep_{new} = \langle e_{e1}, e_{new}, 0, \infty, g \rangle$. We also add a justification of the form $J = \langle e_{e1}, e_{new}, 0, \infty \rangle$. These last two additions ensure that as soon as the guard of the transition is satisfied, A will transition to its next state.

3.1.1 Making Choices with Backtracking Search

At this point we have described the choices and changes tBurton must make to traverse the space of partial plans. We implement tBurton as a backtracking search, but use several additional algorithms to make this search more efficient: a `subplanner` implemented as a wrapper around an 'off-the-shelf' temporal planner, (`UnifyHistories`) to efficiently choose goal orderings, Incremental Temporal Consistency (`ITC`) to efficiently check whether the temporal constraints used in the episodes are temporally consistent, and Causal Graph Synthesis (`CGS`) to simplify traversing the causal graph by reducing the parent-child relation of the causal graph to a sequence of numbers. We summarize the tBurton algorithm in this subsection and then sketch out each of the component algorithms.

In order to maintain search state, tBurton uses a queue to keep track of the partial plans, SP_{part} that it needs to explore. For simplicity, one can assume this queue is LIFO, although in practice a heuristic could be used to sort the queue. To make search more efficient, we make two additional modifications to the SP_{part} we store on the queue. First, we annotate SP_{part} with both the automaton we are currently making choices on, as well as which of the three choices we need to make next. The second involves the use of ITO. When tBurton needs to choose a goal ordering for a given partial plan, it could populate the queue with partial plans representing all the variations on goal ordering. Since storing all of these partial plans would be very memory intensive, we add to the partial plan a data structure from ITO, which allows us to store one partial plan, and poll it for the next consistent variation in goal ordering.

Algorithm 1 provides the pseudo code for tBurton's high-level search algorithm. The algorithm reflects the three choices tBurton makes. In general, backtracking search traverses a decision tree and populates a search queue with all the children

nodes of the expanded node. Since this can be very memory intensive for plan-space search, we take an alternate approach of populating the queue with the next child node to explore as well as a single child node representing 'what not to do' for the remaining children. Line 14 takes advantage of the incremental nature of ITO, to queue the next possible set of goal orderings. Lines 20-21, queue a partial state-plan with a guard representing which value assignments have already been tried. Lines 28-29, queue a state-plan with new goal episodes which remove already tried sub-plans. The behavior of the algorithm with a LIFO queue is to plan for the children automata in the causal graph before their parents, and to plan in a temporally forward manner for each automata.

3.1.2 Subplanner

tBurton's backtracking search exploits the structure of the causal graph, but finding a sub-plan for a factor, an automaton in the causal graph, is still hard. In general, we can leverage the state-of-the-art in temporal planners to find these sub-plans. However, there are a few special sub-plans we can compute without search. When asked to plan for a control variable u, the sub-plan will involve a single value episode where the state constraint is $\perp$ with temporal constraints $lb = 0$, $ub = \infty$. This sub-plan represents our assumption that there is no transition behavior for commands. When asked to plan from one location to another for an automaton A, we can exploit the locality of information from having automata, and check the set of transitions of A. If there is a sequence of automatic timed transitions, we can return the subplan without search. Finally, the subplanner's functional behavior, in which it returns a plan, given an automaton, the initial variable assignment, a goal variable assignment, and a $[lb, ub]$ duration, is easily memoized.

Chapter 6 presents the subplanner in more detail.

Algorithm 1: tBurton(TCA, SP_{part})

Input: Timed Concurrent Automata $TCA = \langle L, \mathcal{C}, \mathcal{U}, \mathcal{A} \rangle$, Partial State Plan SP_{part}
Output: Total State Plan SP_{total}

```
// factor the problem to create the causal graph
```
1 $\{F_1, F_2, ..., F_n\} \leftarrow CausalGraphSynthesis(TCA)$;
```
// add the partial plan to the queue
```
2 F_i is the lowest numbered factor with a goal in SP_{part}
3 $SP_{part} \leftarrow initUnifyHistories(SP_{part}, open\_goals(F_{\{i\}}, SP_{part})$;
4 $Q.push(\langle SP_{part}, F_i, choose\_goal\_order \rangle)$;
5 **while** $Q \neq \emptyset$ **do**
```
    // remove a partial plan from the queue.
```
6 $\langle SP_{part}, F, choice \rangle \leftarrow Q.pop()$;
7 **switch** *choice* **do**
8 **case** *choose_goal_order*
9 **if** SP_{part} *has no open goals* **then**
10 **return** SP_{part};
11 **else**
12 $SP_{part} \leftarrow UnifyHistories(SP_{part})$;
13 **if** SP_{part} *exists* **then**
14 $Q.push(\langle SP_{part}, F, choose\_goal\_order \rangle)$;
15 $Q.push(\langle SP_{part}, F, choose\_value \rangle)$;
16 **case** *choose_value*
17 l or u of F is used in the guard g of open goal episode ep_g
18 **if** $sc \neq false$ **then**
19 Choose assignments $\bigwedge(x = v)$ that entail sc, $x = \{F.l, F.u\}$
```
        // update guard and re-queue
```
20 $SP_{part\_up} = SP_{part}$ with new guard $sc \leftarrow sc \wedge \neg \bigwedge(x = v)$;
21 $Q.push(\langle SP_{part\_up}, F, choose\_value \rangle)$;
```
        // add chosen values to partial state plan
```
22 Add $\bigwedge(x = v)$ as value episodes to SP_{plan}
        $Q.push(\langle SP_{part}, F, choose\_subplan \rangle)$;
23 **case** *choose_subplan*
24 ep_s and ep_e are sequential value episodes of the value history for l or u of F in SP_{part}, separated by a temporal constraint dur.
25 $subplan \leftarrow$ subplanner(F,ep_s.sc,ep_e.sc,dur);
26 **if** *subplan exists* **then**
27 Add *subplan* and parent subgoals to SP_{part}
28 $SP_{part\_up} = SP_{part}$ with negated *subplan* as goal-episodes.
29 $Q.push(\langle SP_{part\_up}, F, choose\_subplan \rangle)$;
30 **if** l *or* u *of* A *is used in open goals of* SP_{part} **then**
31 $Q.push(\langle SP_{part}, F, choose\_value \rangle)$;
32 **else**
33 F_i is the lowest numbered automaton with a goal in SP_{part}
34 $SP_{part} \leftarrow initUnifyHistories(SP_{part}, open\_goals(F_{\{i\}}, SP_{part})$;
35 $Q.push(\langle SP_{part}, F_i, choose\_goal\_order \rangle)$;
36 **return** *failure*;

3.1.3 Incremental Total Order: Unifying Goal Histories

One of the choices that tBurton needs to make is how to order the goals required of an automaton. More specifically, given a set of goal-episodes, we want to add temporal constraints relating the events of those goal-episodes, so any goal-episodes that co-occur have logically consistent guards. By computing the total order over all the goal-episodes we can discover any inconsistencies, where, for example two goal-episodes could not possibly be simultaneously closed. In practice, computing the total order of goal-episodes is faster than computing a plan, so we can discover inconsistencies inherent in the set of goal-episodes faster than we can discover that no plan exists to get from one goal-episode to the next.

Our incremental total order algorithm builds upon work which traverses a tree to enumerate all total orders given a partial order [44]. We modify their algorithm into an incremental one consisting of a data structure maintaining the position in the tree, an initialization function, `init`, and an incremental function, `next`, which returns the next total order.

Our UnifyHistories algorithm is similarly divided into two pieces. The `initUnifyHistories` algorithm creates the Unify data structure when called with SP_{part} and a particular automaton A. `UnifyHistories` can then be called repeatedly to enumerate the next consistent ordering of goal episodes where the location variable l of A is involved.

The UnifyHistories algorithm is described in Chapter 4.

3.1.4 Causal Graph Synthesis and Temporal Consistency

The Causal Graph Synthesis (CGS) algorithm is based on the algorithm used for Burton [57], and is simple to describe. Given a TCA, CGS checks the dependencies of each automaton in TCA and creates a causal graph. If the causal graph contains

cycles, the cycles are collapsed and the product of the participating automata are used to create a new compound automata. Finally, each automaton in the causal graph is numbered sequentially in a depth-first traversal of the causal graph starting from 1 at a leaf. The numbering imposes a search order (with 1 being first), which removes the need to choose which factor to plan for next.

The Incremental Temporal Consistency (ITC) algorithm is used to check whether the partial plan SP_{part} is temporally consistent, or that the temporal constraints in goal, value histories, and justifications are satisfied. Since tBurton will perform this check many times with small variations to the plan, it is important this check be done quickly. For this purpose we use an extension of the incremental temporal consistency algorithm described in [52]. We describe our version of ITC in more detail in Chapter 5.

Chapter 4

History Unification: Efficiently Merging Goal Histories

Recall that tBurton plans by recursively traversing the factors of a causal graph. From a high-level view, the recursive element for a given factor consists of three steps. First, tBurton starts by collecting the goals that factor must achieve in the form of goal-episodes. Second, those goal episodes are ordered into a consistent goal-history through *History Unification.* Finally, planning tells us how to achieve that goal-history. If a plan exists, its subgoals are regressed to a parent factor to repeat the process. (Of course, no search would be complete without backtracking. If no plan exists, the search backtracks, and History Unification is responsible for outputting another consistent goal history. And if no such history exists, the search backtracks to child factors who must replan.)

History Unification is perhaps the most important problem in tBurton because it bridges the gap between factored regression and planning. It allows the plethora of planning techniques which excel at generating a plan for a single initial-state and goal-state, to be applied to a set of time-evolved goals, without concern for where

those goals came from. Towards this end, History Unification must order an input set of goal episodes into goal histories consistent with the existing plan.

There are three important aspects of this output. *Ordering* the goal episodes, allows planners to easily plan between sequential pairs of goals just as they would usually plan between an initial state and a goal. Ensuring the goal history is *consistent* with the existing partial plan decouples the problem of achieving a goal history from interfering with plans for other factors. And, generating *all* achievable goal histories may be necessary if a plan does not exist for a particular goal history.

To see why these are important aspects, take the simple example of picking up one's children after school. We need to pickup 'Amy from school' and 'Brad from soccer' but have a certain time-limit in which to do so (lest be fined by the school district). In order to start planning, we need to know which goal should be achieved first. While we could arbitrarily choose an order, it makes more sense to first consider these goals in the context of our children's plans. If we know Amy gets out before Brad, we can order our goals consistently by picking-up Amy before Brad. On the other hand, if they let out at the same time, the pickup order might depend more on our knowledge of local traffic. To determine which pickup order is achievable, we could try planning to pick-up Amy then Brad and if that is discovered to be unachievable, Brad then Amy.

History Unification is therefore more than just an ordering problem. The manner in which candidate goal histories are created dictates the branches explored when searching for a plan. In a sense, the History Unification problem can be thought of as the generator in a generate-and-test problem formulation. History Unification generates a goal-history, a planner checks to see if that goal history is achievable through a plan, and if not achievable History Unification is responsible for generating another goal history.

Beyond the basic requirements of History Unification, we can also make the pro-

cess of enumerating all achievable goal orderings more efficient. The algorithm we describe in this chapter will additionally: check that a goal episode ordering is self-consistent before returning it, thus filtering any obvious inconsistencies such as being both at school and soccer at the same time; compute goal histories incrementally, thus avoiding the time and space penalty of calculating and storing all of the possibilities; and will prune inconsistencies early by learning from conflicts. If we discover we must pickup Amy before Brad we want to ensure that ordering is maintained in all subsequently created goal histories, no matter how many additional goals we may have to juggle.

Before proceeding with a more in-depth discussion of our approach to History Unification, it is worth noting that the approach we present in this chapter is quite general. Our algorithm for History Unification can be used to efficiently find consistent goal episode orderings from a set of goal-episodes, regardless of how that set was formed. [1]

For tBurton, however, the set of goal episodes ordered by any instance of the History Unification problem only concerns a single factor and can only come from two sources. In the recursive case, the goal episodes we are interested in ordering come from child factors. In the base case, goal episodes can also come from the 'user' supplied initial partial plan. This application of History Unification by factor is an important ramification of how tBurton is structured, and is designed to take advantage of the locality of information gained by factoring. Since factoring has already allowed us to separate the goal episodes concerning one factor from the goal

[1] In the most general sense, History Unification is the problem of systematically checking the consistency of constraints described over a timeline. In the development of our algorithm to solve the unification problem, we will describe temporal and state-constraint consistency as subproblems, but single-out planning as a special algorithm. This exposition suits the overall planning story. However, planning can also be considered a consistency checker for reachability between goals. Thus temporal consistency checking, state-constraint consistency checking, and invoking a subplanner, can all be considered sub-problems of History Unification.

episodes of another, History Unification can focus on ordering only the goal episodes of a particular factor.

In this chapter we present *UnifyHistories*, a *conflict-directed* History Unification algorithm that can take a set of goal episodes and *incrementally* output *consistent* goal histories.

Since a goal episode can be thought of as a pairing between a state-constraint (that describes what must be true) and a temporal constraint (that describes when it must be true), there are two notions of consistency we can establish. A goal history is consistent if its ordering of goal episodes is both *temporally consistent* and *state-constraint consistent.* Temporal consistency asks whether a goal history is schedulable when the ordering it imposes over its goal-episodes is considered with regards to the rest of the partial plan. State-constraint consistency asks whether any 'overlapping' goals are trivially unachievable, and can be thought of as a form of self-consistency check within a goal-history. For example: Is it possible to 'have cake' and 'eat-cake' at the same time? Is it possible to both be at school and soccer to pickup my children at the same time?

By incremental, we mean an algorithm of which we can ask 'on-demand' for the next consistent ordering, thus avoiding the need to clutter tBurton's queue with all-possible goal orderings. In our approach to History Unification, candidate goal histories are generated one at a time, creating all consistent overlapping orderings of goal episodes in the limit.

By conflict-directed, we mean an algorithm that can adapt to new information. During the planning process, we will discover that certain goal orderings are impossible because no plan can reach from one goal-episode to the next. In essence, planning helps discover constraints that History Unification cannot discover alone, because they are imposed by the model. History Unification can 'learn' these constraints and take them into account when computing the next goal ordering.

There are four motivations for this approach: one concerns the importance of establishing the consistency, and the others are about the use of ordering. First, by establishing a consistent goal history before planning, we can save otherwise wasted effort planning for unachievable goals. Second, ordering the goal-episodes provides a systematic way to test for consistency. Third, for factors consisting of only a single automaton, some ordering is required, because it would be impossible for an automaton to be in two locations at one time. Fourth, the ordering of goals episodes divides the planning problem into smaller pieces. By asking a planner to generate plans from one goal to the next, we can use the planner to not only generate a plan, but learn from its failures to prune future goal orderings.

In this chapter we present the details behind *UnifyHistories*, an algorithm designed to incrementally produce consistent goal-episode orderings. We start with an overview of our approach to the History Unification problem with a more formal treatment. We then proceed to solve the History Unification problem by building upon *FindAllLinearExtensions*, an algorithm that efficiently generates all total orders from a partial order. Section 4.3 presents *FindAllLinearExtensions* and the theory behind it. The subsequent sections elaborate on this total-ordering algorithm to develop *UnifyHistories-v2*, our incremental, consistent, and conflict-directed answer to History Unification. Section 4.4 describes *Incremental Total Order* (*ITO*), a modification to *FindAllLinearExtensions* that can incrementally generate all total orders. Section 4.5 describes *UnifyHistories-v1*, in which ITO is applied to the problem of History Unification. *UnifyHistories-v1* can incrementally generate all self-consistent goal histories, making it the first algorithm that satisfies the basic requirements of History Unification. Finally, Section 4.6 describes the conflict-directed *UnifyHistories-v2*, where invalid orderings discovered during planning are learned and used to prune goal histories.

4.1 History Unification

In this section we formalize the problem of History Unification. We start with an example which we will use to clarify these ideas.

4.1.1 Example

At the start of this chapter we used a simple example involving picking up our children to illustrate the importance of generating more than one consistent goal ordering. We now return to and explicate this example.

In our extended example, there are four automata, two automata for the children, Amy and Brad, one for the Car that will pick them up, and one for the Dad that drives it. Figure 4-1 shows the causal graph involving these four automata. Since History Unification leaves the details of traversing automata to planners, the figure abstracts away the details of the automata into orange blocks. The remaining orange arrows show the causal edges. For this example, the children depend on the Car to pick them up, but the Dad and the Car are cyclically dependent. The Dad needs the Car to get around and the Dad is needed to drive the Car. To enforce the acyclicity of the causal graph, Amy and Brad are their own factors, while the Car and Dad are joined in a single factor.

At this point in the planning process, three goals remain. The plans for our child factors have been determined, and they have each regressed goal-episodes to the parent factor. Amy needs to be picked up from school, thus she requires the Car be at school during a specific time. Similarly, Brad needs to picked up from soccer practice and requires the Car be at soccer during a specific time. Figure 4-2 shows the partial plan. Ellipses are used to hide details of the plan that are unimportant. To make the problem a little more interesting, we have also added the tangential goal the Dad has to listen to his audio book.

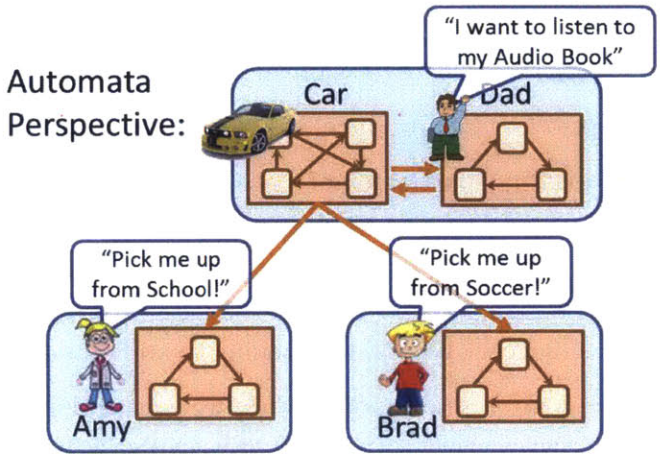


Figure 4-1: Causal graph for the pick-up children example. Orange squares represent the four automata: Amy, Brad, Car, and Parent. They are joined by bright orange causal edges. Factors are enclosed in blue rounded rectangles. The goals of Amy, Brad, and Dad are expressed colloquially in thought bubbles.

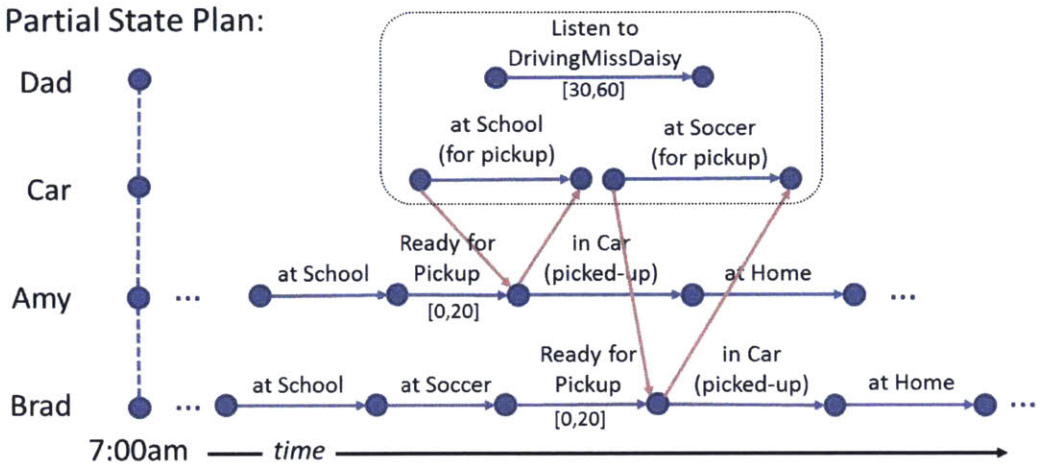


Figure 4-2: Partial state plan for the pick-up children example. At this point in the planning process, Amy and Brad's day are set. The remaining goals have Amy and Brad being picked up while the Dad listens to an Audio Book. The red arrows indicate that Amy and Brad's request to be picked up come with temporal constraints. The parent must be aware of those constraints when deciding whether to be at school or soccer first.

4.1.2 The History Unification Problem

As the bridge between factored regression and planning, History Unification must take a set of goal episodes and produce an ordering of those goal episodes that is consistent with the existing partial plan, potentially producing all such consistent orderings.

For our example problem, History Unification takes the goal-episodes representing 'Car at school', 'Car at soccer', and 'Dad listening to audio book' as input. But, what does it mean to order goal-episodes, and how do we ensure that ordering is consistent with the rest of the plan?

Why Order?

The basic building block of tBurton's state-plan representation is an *episode*, which asserts a state-constraint over the duration of a temporal constraint. But, beyond being asserted during that duration, the state-constraint is not a function of time.

History Unification exploits this weak coupling between temporal constraints and state-constraints. Ordering is *nearly* sufficient to decouple the problem of checking for temporal consistency from the problem of reasoning over the state-constraints.

For a state-constraint which is asserted during a corresponding temporal-constraint, but is otherwise not a function of time, ordering is sufficient to tell us which state-constraints will be simultaneously asserted. For example, if Dad starts listening to his audio book before picking up his children and stops listening afterwards, the time and duration with which each goal is achieved is not needed to tell us which goals this plan must achieve simultaneously. The ordering information is sufficient to tell us that we need to check whether simultaneous listening and pick-up is possible.

For planning, which can be viewed as a form of reachability checking between sequential state-constraints, ordering tells us which state-constraint (or simultaneously

asserted set of state-constraints) needs to be achieved before the other.

Note that we said 'ordering' is nearly sufficient to decouple reasoning over temporal and state-constraints. This is because the question of reachability answered by a temporal planner not only involves a pair of sequential state-constraints, but the time allowed to get from one state-constraint to the next. This remaining coupling is not difficult to resolve, as ordering allows us to extract the temporal constraint between state-constraints and pose it as part of the planning problem given to the planner.

How to Order?

Ordering a set of goal episodes is not always as straight forward as putting one goal entirely before the other. Take for example the two goals, 'pat head' and 'rub tummy'. These two goals are not mutually exclusive. To allow their co-occurrence, it is important that the goal orderings produced by History Unification also include 'overlapping' orderings. These orderings are identical to that of Allen's interval algebra [1]. Goal episodes can temporally 'overlap', they can share a start and/or end event, or they can completely preceded or follow each other.

The need to support overlapping orderings occurs in two cases in the context of the factored problem representation used by tBurton. Consider the case where a factor consists of only one automaton. Since a TCA Automaton can only be in one location at any give time, it would seem that overlapping goal orderings should be disallowed. But, this assumes goal episodes always assert different state-constraints. In a factored space, where goal episodes are being regressed from different sources, goal episodes could assert the same state-constraints over different intervals of time. Take, for example the process of putting a child to bed, if one child factor requires its parent to 'read a story' for 10 minutes, and a sibling factor requires 'read a story' for

12 minutes, it may be more efficient for the make-span of the plan or even necessary for the temporal feasibility of the child plans to allow these goals to overlap.

The second case for overlapping orderings occurs when the goal episodes have been regressed to a factor consisting of multiple automata. Since the automata could transition simultaneously, it should be obvious that overlapping goal orderings need to be considered to allow their asynchronous behavior. In our pick-up children example, the Dad and Car automata are grouped into one factor, but since these are separate automata, the goal the Dad has of listening to an audio book can happen concurrently with picking up his children.

In general, two goal-episodes should be allowed to co-occur when their state-constraints could be achieved simultaneously.

Establishing Consistency with the Partial Plan

Once an ordering has been established, we need to make sure that ordering is consistent with the partial plan. If we know a particular goal ordering is consistent with the rest of the plan, then we can plan for that ordering in isolation, without considering the rest of the plan.

Since tBurton requires that all goal episodes be regressed to a factor before solving the History Unification problem, there are no state-constraints coupling us to other factors. Only temporal constraints remain. By the nature of temporal constraints, checking for consistency means we must consider all of the temporal constraints that underly the partial plan. But, to give this temporal consistency problem more perspective, we can classify these temporal constraints by their purpose. We need to consider the temporal constraints that:

- are used to specify the durations of the goal episodes,
- are implicit in the ordering of the start and end events of the goal episodes,

- justify the goal episode in relation to the factored plans from which they were regressed, and
- participate in the plans, goal episodes, and justifications of other factors.

There are many ways to check for temporal consistency. In the next chapter, Chapter 5, we present Incremental Temporal Consistency (ITC), an incremental and efficient way to check for temporal consistency.

4.1.3 Problem Formulation

Input: State Plan and Goal Episodes

The History Unification problem takes as input, a tuple $\langle SP, EP_G \rangle$.

Recall that a state plan, SP, is a collection of events and episodes classified by purpose as a goal-history GH, a set of value histories $\mathcal{VH}$, and a set of justifications $\mathcal{J}$ (Definition 4). History Unification needs the entire state-plan so it can check the temporal consistency of a particular goal order in the context of the entire state plan.

EP_G is the subset of the episodes from the goal-history, GH, that History Unification must consistently order. More formally, for a goal history GH consisting of events EV_{GH} and episodes EP_{GH} (Definition 3), $EP_G \subseteq EP_{GH}$.

An element of EP_G is referred to as a *goal-episode*, and takes the form $ep = \langle e_s, e_e, lb, ub, sc \rangle$ (Definition 3). e_s and e_e are referred to as the *start* and *end* events of the goal episode, respectively. The components e_e, e_s, lb, ub define a *simple temporal constraint* $lb \leq e_e - e_s \leq ub$, while sc is a propositional state-logic formula that expresses a set of desired state assignments that must be maintained during the duration allowed by the simple temporal constraint.

Using the definition of a goal-episode, we can define EV_G, as the set of start and end events of the episodes in EP_G, $EV_G = \bigcup_{ep \in EP_G} \{ep.e_s, ep.e_e\}$. Consequently, for

$GH = \langle EV_{GH}, EP_{GH} \rangle$, EV_G is a subset of EV_{GH}, just as EP_G is a subset of EP_{GH}.

While we have informally described History Unification as "ordering goal-episodes", the set EV_G is very useful because it denotes the actual elements we are interested in ordering, the start and end events of the goal-episodes.

Output: Ordered Goal Episodes

The output of History Unification is SP', a modified version of the input state-plan SP in which temporal constraints, in the form of episodes, have been added to enforce the ordering over the events EV_G such that the set of goal-episodes EP_G is *consistently ordered.* We first define the two types of goal-episodes that can be add to the goal-history, and then return to define '*consistently ordered*'.

Definition 10 (Sequence Episode). A *sequence episode*, $ep_{sequence} = \langle e_i, e_j, 0, \infty, true \rangle$, requires that event e_i precedes e_j, where e_i and e_j are events in EV_G.

Definition 11 (Co-occur Episode). A *co-occur episode*, $ep_{co\text{-}occur} = \langle e_i, e_j, 0, 0, true \rangle$, requires that event e_i and e_j occur at the same time, e_i, $e_j \in EV_G$. (The temporal constraint that a co-occur episode represents is sometimes also referred to as a 'zero-relation' because of its 0-valued lower and upper bounds.)

The modifications that result in SP' can be formally defined relative to the input state-plan SP and events EV_G. Let the goal history of SP' be $\mathcal{GH}' = \langle EV'_{GH}, EP'_{GH} \rangle$ and the goal history of SP be $\mathcal{GH} = \langle EV'_{GH}, EP'_{GH} \rangle$. History Unification need only add episodes to the goal history, therefore $EP'_{GH} = EP_{GH} \cup EP_{order}$, where EP_{order} is a set of sequence and co-occur episodes defined to only constrain events in EV_G. The goal-events, value histories, and justifications of SP remain unaltered in SP'.

Of course, History Unification cannot arbitrarily add sequence and co-occur episodes. They must be added in such a way to ensure the goal-episodes in SP' are *consistently*

ordered. We touched upon the notion of consistently ordered at the start of this section, we now return to define 'goal-episode order' and what it means for that order to be 'consistent'.

The idea of an 'order' within a plan requires some understanding of what it means to schedule a plan.

Definition 12 (Schedule). A *schedule* is a function, $s : EV \rightarrow \mathbb{R}$, that assigns real-valued times to events.

Definition 13 (Temporally Consistent). A state plan is *temporally consistent* if a schedule exists for all of the events in the state plan that satisfies all the temporal constraints in the state plan. Equivalently, we will sometimes say a schedule is *temporally consistent*, when the source of temporal constraints is clear (i.e. with respect to a state plan).

Informally, a set of goal-episodes EP_G is *ordered* when, for all temporally consistent schedules of SP', the order of execution of the events in EV_G does not change.

Definition 14 (Goal-Episode Order). Let schedule s and s' be two temporally consistent schedules for $\langle EV_G, EP_G \rangle$ and e_i e_j be two events in EV_G where $i \neq j$. A set of goal-episodes EP_G is *ordered* when, $s(e_i) \leq s(e_j)$ iff $s'(e_i) \leq s'(e_j)$ for all schedules s,s' and all events e_i,e_j. (A set of goal-episodes is considered trivially ordered if there is only one temporally consistent schedule).

Definition 15 (Consistent Goal-Episode Order). An ordering of goal-episodes EP_G is *consistent* if SP' is *temporally consistent* and the ordering of EP_G is *state-constraint consistent.*

Definition 16 (State-Constraint Consistent). An order of goal-episodes EP_G is *state-constraint consistent* if the conjunction of state-constraints from temporally overlapping episodes is not `false`. Let $t \in \mathbb{R}$ be time, s be a schedule of events EV_G,

and $ep \in EP_G$ take the form $ep = \{e_i, e_j, lb, ub, sc\}$. An ordering is state-constraint consistent if, for all $t \bigwedge_{ep \in EP_G} (s(ep.e_i) < t \leq s(ep.e_j) \rightarrow ep.sc)$.

For this chapter, we will use the term *consistent* as a convenient abbreviation for '*consistent ordering of goal episodes*' (Definition 15), and qualify the participating notions of *temporal* consistency and *state-constraint* consistency whenever we only require a particular notion of consistency.

4.2 Our Approach to History Unification

Our approach to History Unification builds upon an algorithm for generating total orders [44], which we will refer to by its original published name `FindAllLinearExtensions`.

`FindAllLinearExtensions` is capable of efficiently generating all total orders given a partially ordered set. In particular, there are a number of features that make `FindAllLinearExtensions` a good base for our approach to History Unification:

- It can generate all total orders, making it easy to demonstrate the completeness of our approach.
- The generation procedure is predictable and systematic, making it easy and efficient to prune subsets of total orders.
- The generation of any total order can occur in O(1) time.
- It uses a small memory footprint, linear in the number of elements in the set.

4.2.1 The Trouble with Ordering

As we have already discussed, ordering goal episodes is critical to History Unification. But, there is a mismatch between the types of orders we can get from `FindAllLinearExtensions` and the types of goal-episode orders we need.

One way to apply `FindAllLinearExtensions` is to use the set of goal episodes as the partially ordered set, thus interpreting each goal episode as an atomic unit. In this application, `FindAllLinearExtensions` would generate all *non-overlapping* orders of goal episodes. Since we need *overlapping* goal episodes, we have to do something different.

Instead, we use the set of events that define the goal episodes as the elements of the partially ordered set. For two goal episodes, AB and CD, `FindAllLinearExtensions` will generate 6 possible orderings (Figure 4-3).

These 6 orderings are a subset of the 13 orderings we need to generate to solve the History Unification problem. Figure 4-4 depicts the 13 expected orderings, as defined in Allen's interval algebra [1]. Highlighted in red are the 7 missed orderings. These 7 orderings are missed because they have co-occurring events.

To ensure we can generate these 7 missing goal episode orderings, we will use one additional rule: If the conjunction of constraints between two sequential events is false and the two sequential events do not appear in the same goal episode, the two events can be joined to produce a new ordering. The two conditions of this rule can be explained intuitively.

4.2.2 Establishing Consistency

In the process of generating a total order, UnifyHistories also checks for both the temporal consistency of the ordering and the consistency of co-occurring state-constraints (goal-episodes that overlap in time, which may require competing goals). Temporal consistency is checked by calling Incremental Temporal Consistency (ITC) (Chapter 5) for each goal-episode ordering, but considers the order in the context of the entire partial-plan to ensure all metric temporal constraints are observed. Figure 4-5 depicts a set of goal-episodes that have been totally ordered (dotted arrows) and the justifica-

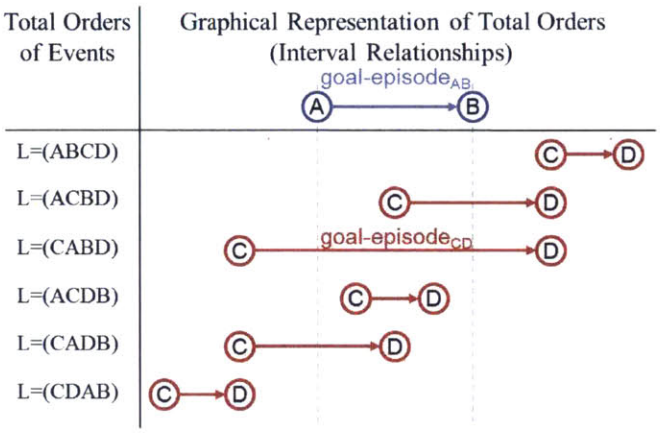


Figure 4-3: The six possible goal-episode orderings UnifyHistories considers for a pair of goal-episodes.

tions (orange) that temporally constrain their occurrence relative to other value and goal episodes. Even though UnifyHistories only orders goal episodes, it must consider the temporal consistency of that ordering in the context of the temporal constraints of the entire plan.

State-constraint consistency is checked by traversing the goal-episode ordering using an incremental SAT solver to check whether any co-occurring state-constraints are trivially un-achievable (resolve to false). Figure 4-6 depicts the same set of totally ordered goal-episodes as figure 4-5, but this time focuses on their corresponding state-constraints. By taking the conjunction of state-constraints from overlapping episodes, we can produce an equivalent (when considering only state-constraints) projected sequence of goal-episodes. In this example, UnifyHistories has discovered that the overlap of goal-episodes AB, CD, and EF is infeasible because the conjunction of their state constraints: $(A = 1 \vee A = 2) \wedge (A = 1 \vee B = 2) \wedge (A = 3 \wedge B = 2)$ is `false`.

Even though the projected goal episodes have `false` for several state constraints, this goal-episode ordering may still be correctable. In this case UnifyHistories will

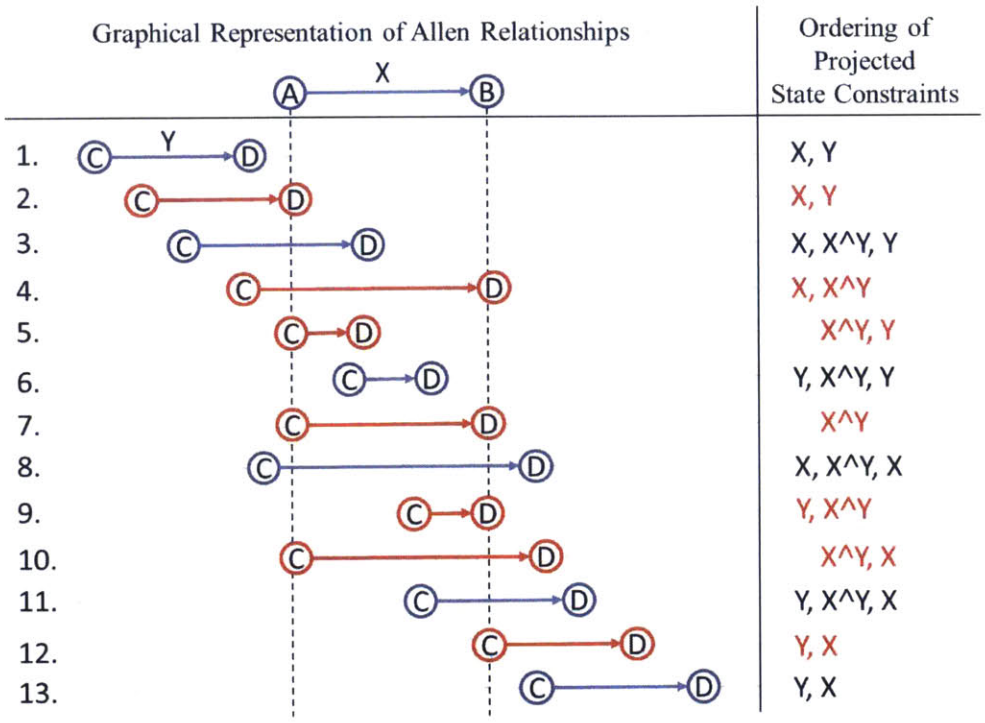


Figure 4-4: A graphical representation of the 13 Allen interval relationships for two goal episodes A-B and C-D. Their corresponding state-constraints are symbolically represented with X and Y. The sequences of projected state-constraints are shown in the right hand column. The relationships not created by `FindAllLinearExtensions` are highlighted in red.

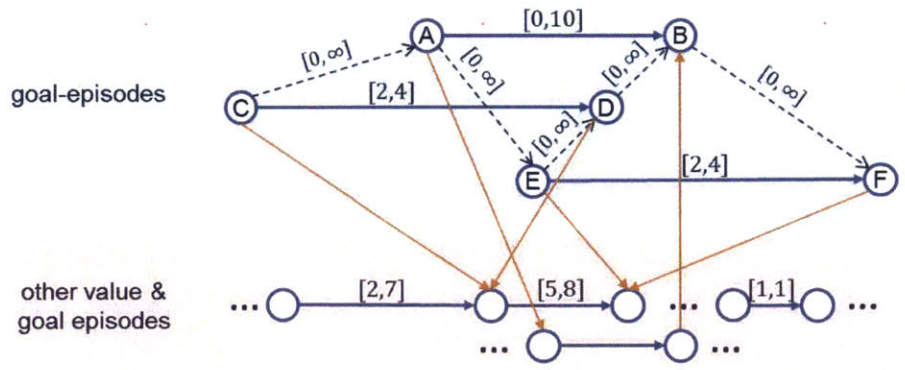


Figure 4-5: An ordering of goal episodes must be temporally consistent relative to the partial plan. The ordering is depicted as dotted arrows. The orange arrows represent temporal constraints relating the goal episodes to the rest of the partial plan. For brevity, the values of some temporal constraints are not depicted.

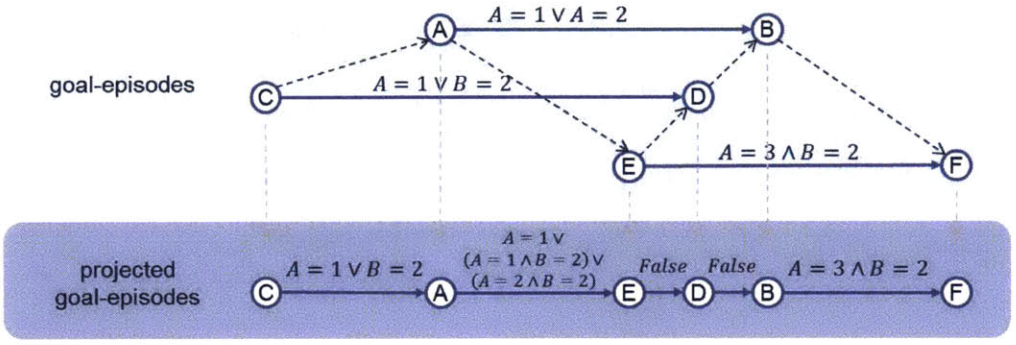


Figure 4-6: An ordering of goal episodes must have consistent overlapping state-constraints, unlike this particular ordering f goal episodes.

check the temporal consistency of requiring events E, D, and C co-occur, effectively removing the overlap that resulted in the inconsistency. This collapsing of constraints allows us to recover all of the Allen Relationships.

In order to create the 6 goal-episode orderings for all pairs of episodes that Unify-Histories must consider, we will totally order the start and end events of each episode. In the next section, we present this ordering algorithm.

4.3 Generating Linear Extensions

A total order is also sometimes referred to as a linear-extension. The algorithm at the core of our approach to History Unification is a constant-time linear extension generator by Ono and Nakano from 2005 [44]. Their algorithm takes a partial order, and once initialized (which is not necessarily a constant-time operation), generates all possible total-orders, with each total order output in O(1) time.

We start by defining a few important terms, using the same notation as the original paper, and then proceed to describe the algorithm.

4.3.1 Notation

Definition 17 (Partial Ordered Set)**.** A *partial ordered set*, or *poset* $\mathcal{P}$, is a set S with a binary relation R, which is reflexive, antisymmetric and transitive. The relation R is sometimes referred to as the *partial-order* on S. We use $|\mathcal{P}|$ or n to denote the size of a poset: the number of elements in S.

Definition 18 (Linear Extension)**.** A *linear extension* of $\mathcal{P}$ is a sequence containing all elements of set S, in which each adjacent pair in the sequence is related by R.

For example, a poset $\mathcal{P}$ with set $S = \{a, b, c, d, e, f\}$ and relations $R = \{(a, b)(b, c)(d, e)(e, f)\}$ admits the linear extensions: $L = (d, e, f, a, b, c)$ and $L =$

(d, a, e, b, f, c), among others.

Their algorithm is simple, but ingenious, and depends on the insight that it is possible to define a tree of all linear extensions of a poset. The tree can then be walked to output each extension. The tree is defined as follows.

Given a poset $\mathcal{P}$, choose any linear extension $L \in LE(\mathcal{P})$. Let $L = (x_1, x_2, ..., x_n)$ be this linear extension. For simplicity, and without loss of generality, we will refer to the elements of this poset by their indexical numbering, $x_i = i$, such that $L = (1, 2, ..., n)$. To define a tree of these linear-extensions, they introduce the notion of a *level.*

Definition 19 (Level). A *level* is the number that should appear in place of the first non-sequential number in the linear extension.

By this definition, the linear extension $L = \{1, 2, 4, 3\}$ has a level of 3.

Together, a linear extension and its level define a node in a tree of linear extensions. To give the tree structure, we must also define the parent relationships between these nodes.

Definition 20 (Parent). The *parent* of a linear extension $P(L)$ is a linear extension L' with a larger level, $level(L') = level(L) + 1$, in which all elements of L and L', ignoring the $level(L)$ element, have the same total order.

For example, $L' = (1324)$ is the parent of $L = (3124)$, $L = (3214)$, and $L = (3241)$ because $level(L') = 2$, $level(L) = 1$, and without the element 1, each linear extension has (324) as component orders.

Figure 4-7 shows the tree of linear extensions for the poset $\mathcal{P}(\{1, 2, 3, 4\}, \emptyset)$ (a poset with no partial-order relation), rooted at the linear extension $L(\mathcal{P}) = (1234)$. The level of each linear extension is emphasized by an underline. For example, the level of L=(2$\underline{1}$34) is 1 because 1 is not in its proper place as the first element. The root of the tree is the linear extension $L = \{1234\}$, which is said to have level $n + 1$.

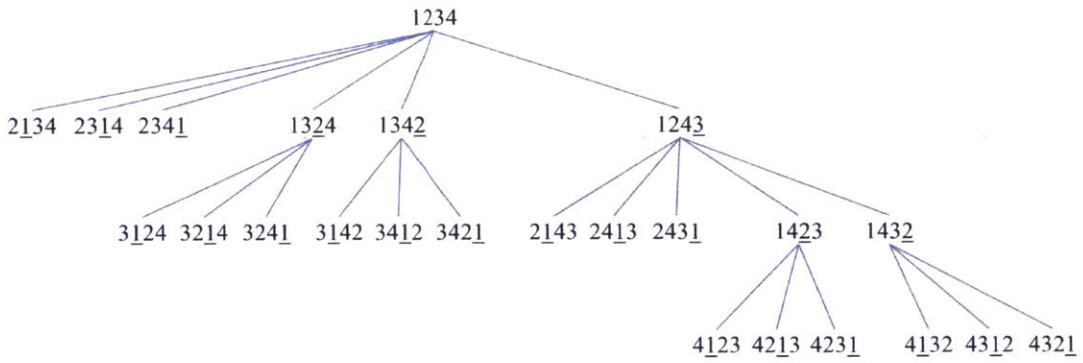


Figure 4-7: The tree of linear extensions for the set $\{1, 2, 3, 4\}$ with no partial-order relation.

While not critical to the correctness of their algorithm, we take a moment to elaborate on the use of $n + 1$ as the level of the root linear extension. For a poset with n elements, the use of $n+1$ for the root is consistent with the definition of level, but also overly conservative. Recall that the definition of level (Def. 19) is the index of the first element out of place. While it is true that in the root linear extension, no elements are considered out of place, it is also true that the last element could never be out of place. For example, the 'correct' level of $L = (1234)$ is 5, but it can be said to have level 4 because 4 could never be in a higher index.

4.3.2 Algorithm for Generating All Total Orders.

The basic linear extension generation algorithm consists of two parts. Algorithm 2, `FindAllLinearExtensions`, is the entry point to their algorithm, which emphasizes the need to start with one linear extension. Algorithm 3, `FindAllChildren`, does the hard work of walking the tree of linear extensions, rooted at the first extension.

Given a poset, `FindAllLinearExtensions` (Algorithm 2) must first find one linear extension, which will act as the root of the tree of linear extensions (line 1). The core algorithm, `FindAllChildren`, generates the children linear extensions from this root

Algorithm 2: FindAllLinearExtensions

Input: $\mathcal{P} = (S, R)$
Output: void

```
1 Find a linear extension L_r;
2 FindAllChildren(L_r,n);
```

Algorithm 3: FindAllChildren

Input: L=$(p_1 p_2 ... p_n)$,l_p //{L is the current linear extension of $\mathcal{P}$}
Output: void

```
1 Output L;
2 for i = 1 to l_p -1 do
    // generate children with level i
3   j := i;
4   while j < n and (p_j, p_{j+1}) ∉ R do
5     swap p_j and p_{j+1};
6     FindAllChildren(L=(p_1 p_2 ... p_n),i);
7     j := j+1;
8   insert p_j immediately after p_{i-1};
    // now p_i = i again holds, and L is restored as it was
```

(line 2). As discussed in the previous subsection, the version of the algorithm we present here uses level n for the root, instead of the originally ascribed $n + 1$.

`FindAllChildren` (Algorithm 3) consists of a pair of nested loops which result in a recursive call to itself. `FindAllChildren` starts by outputting the linear extension it was given as an argument (line 1). The outer loop selects the level of the immediate children (lines 2-3), while the inner loop creates permutations of the children at a particular level (lines 4-7).

The conditional expression guarding the inner loop (line 4) warrants some additional discussion. The guard $(p_j, p_{j+1}) \notin R$ prunes child linear extensions from violating the partial-order relations defined in the poset $\mathcal{P}$. For ITO, we will replace this guard with a more elaborate function in order to check for various ordering relationships that are hard to express as a simple poset. We also make a minor correction to the guard, by adding the expression $j < n$ to the inner loop condition (line 4). This ensures the index variable j stays within the legal range: the elements of the linear extension.

An additional minor correction we could make is the addition of R, the partial-order relation, to the parameters of the `FindAllChildren` algorithm. R is necessary because it is used by the inner-loop guard. In the provided pseudo-code it is implied that R inherits from the poset used in initial invocation to `FindAllLinearExtensions` and never changes. We do not add this parameter because it confuses a subsequent description of an incremental version of this algorithm.

Figure 4-8 depicts the poset $\mathcal{P}(\{1,2,3,4\}, \{(1,3),(2,4)\})$. Figure 4-9 depicts the corresponding tree of linear extensions, as it is traversed by the `FindAllChildren` algorithm. i and j correspond to the outer (level) and inner (swap permutations) loop variables, respectively. The red 'cancellation' icon indicates that the marked child linear extension was deemed by the inner-loop guard to be in violation of the the poset and was therefore not generated. The red lines show the other child linear

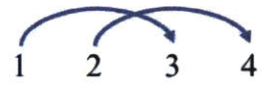


Figure 4-8: The poset $\mathcal{P}(\{1,2,3,4\},\{(1,3),(2,4)\})$.

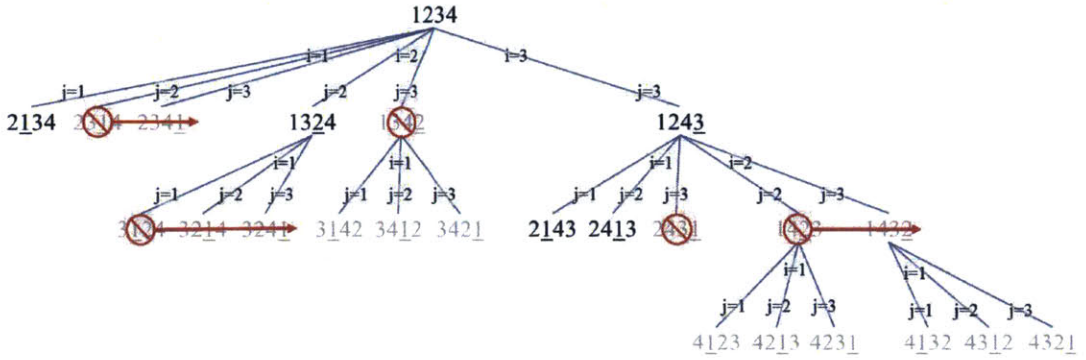


Figure 4-9: The tree of linear extensions for the poset $\mathcal{P}(\{1,2,3,4\},\{(1,3),(2,4)\})$, depicted in figure 4-8

extensions that were not generated as a consequence of the guard exiting the inner loop. We will take advantage of this efficient pruning behavior in UnifyHistories.

Even though `FindAllLinearExtensions` is incredibly efficient at generating total orders, it is written to generate all total orders in one call. In order to get the incremental behavior desired of UnifyHistories, we need to re-factor this algorithm so it only produces one total order per call. In the next section we present *Incremental Total Order* (`ITO`), the incremental version of `FindAllLinearExtensions`.

4.4 Incremental Total Order, Generating Linear Extensions Incrementally

To make `FindAllChildren` incremental, we need to maintain its state across invocations. This can be done by making the program stack, which is maintained

automatically during recursive calls, explicit.

We define this state to be a tuple $ST = \langle R, \mathcal{S}, firstrun? \rangle$. R is the partial order relation. $\mathcal{S}$ is a stack (a LIFO queue), with elements of the form $s = \langle L, l_p, i, j \rangle$. And, $firstrun?$ is a boolean valued variable which indicates whether this is the first run of incremental linear extension. The purpose of $firstrun?$ will become apparent when we present the pseudo code for the incremental algorithm. For now, we will focus on the stack $\mathcal{S}$ and its elements, s.

We interpret s as a stack frame which indicates the next branch of the tree to explore. L and l_p have already been defined as a linear extension and its level, and correspond to the parameters of the algorithm `FindAllChildren`. i and j are its internal loop variables. In relation to a stack frame, i and j act as program counters, keeping track of where we are in the execution of the body of `FindAllChildren`.

Algorithm 4: initITO

Input: $\mathcal{P} = (S, R)$
Output: void
1 Find a linear extension L_r;
2 $ST = \langle R, \mathcal{S}, firstrun? = true \rangle$;
3 $\mathcal{S}$.push($\langle L_r, n, 0, n \rangle$);

Algorithms 4 and 5 provide the pseudo-code for our incremental version of enumerating the linear extensions, which parallel the non-incremental versions presented earlier (Algorithms 2 and 3, respectively).

`initITO` (Algorithm 4), initializes our stack based state, ST by finding one linear extension and adding the corresponding stack frame to ST. It parallels `FindAllLinearExtensions` (Algorithm 2).

`nextITO` (Algorithm 5), takes the state ST and returns one child linear extension per invocation, or null if no such extension exists. This algorithm starts by popping one element from the state's stack (line 3) and then either: generates all linear ex-

Algorithm 5: nextITO

Input: $ST = \langle R, \mathcal{S}, firstrun? \rangle$

Output: a linear extension, L, or `null` if no next extension exists

```
1  if S.isempty() then
2      return null;
3  ⟨L, l_p, i, j⟩ ← S.pop(); // L=(p_1, p_2, ..., p_n)
4  if firstrun? then
5      firstrun? = false;
6      S.push(⟨L, l_p, 0, n⟩);
7      return L;
8  if j < n and (p_j, p_{j+1}) ∉ R then
9      swap p_j and p_{j+1} in L;
10     j=j+1;
11     S.push(⟨L, l_p, i, j⟩);
12     S.push(⟨L, i, 0, n⟩);
13     return L;
14 else
15     if i > 0 then
16         insert p_j immediately after p_{i-1} in L;
           // Now p_i = i again holds, and L is restored as it was
17     i=i+1;
18     if i < l_p then
19         j=i;
20         S.push(⟨L, l_p, i, j⟩);
21     return nextITO(ST);
```

tensions of children of the same level (lines 8-12), or moves to the next level (lines 16-20).

In the midst of this pseudo-code, the state value *firstrun?* is used to guards a few statements (lines 4-7). The purpose of *firstrun?* becomes apparent when thinking about the behavior of the non-incremental `FindAllChildren` algorithm (Algorithm 3). In `FindAllChildren`, the linear extension passed as an argument into `FindAllChildren` is output first and is then followed by code to compute to the next linear extension. In effect, `FindAllChildren` outputs the work done by a previous invocation before setting up an output for the next invocation. This behavior is backwards from what one might normally expect. *firstrun?* is used by the incremental version of linear extension generation to re-invert this behavior, allowing one invocation of `nextITO` to compute a linear extension and output it. By checking whether *firstrun?* is `true` on line 4 of `nextITO` (Algorithm 5), we can output the root linear extension, initially found by line 1 of `initITO` (Algorithm 4). We can also re-queue the initial stack frame (Algorithm 5, line 6), allowing that information to be used on the next invocation of `nextITO` to generate another linear extension.

With this incremental version of linear extension generation, we now have the basis for UnifyHistories.

4.5 UnifyHistories-v1

In this section we present a basic version of UnifyHistories.

The pseudo-code for UnifyHistories-v1 is provided in algorithms 6-11. `initUnifyHistories-v1` (Algorithm 6) initializes UnifyHistory's state, while separate invocations of `UnifyHistory-v1` (Algorithm 9) return different consistent goal-episode orderings. The structure of these algorithms parallels the algorithms from incremental linear extension generation, `initITO` (Algorithm 4) and `nextITO` (Al-

gorithm 5), respectively. The remaining algorithms incrementally maintain SC and EP_{order}, data structures important to the efficient operation of UnifyHistories-v1.

UnifyHistory's internal state $ST = \langle R, \mathcal{S}, firstrun?, EP_G, SC, EP_{order} \rangle$, is an extension of the state used in incremental linear extension generation. The additions are: EP_G, the set of goal-episodes that `UnifyHistories-v1` must order, saved from the inputs of `initUnifyHistories-v1`. SC, an array of projected state-constraints, like the depiction of the state-constraints on the projected goal-episodes of Figure 4-6. And, EP_{order}, a set of ordering episodes added to the state-plan to induce the goal episode ordering computed by Unification.

`initUnifyHistories-v1` (Algorithm 6) starts by creating a poset from the goal-episodes EP_G (lines 1-7). In creating the poset, all start and end events of the goal-episodes are considered part of the set, but only goal-episodes that admit either a strictly positive or negative duration between its start and end events are mapped to relations. This is because only these types of temporal constraints provide the reflexive and antisymmetric relation required by linear extension. Otherwise, we assume that goal-episodes will never have zero-duration.

After the poset is established, `initUnifyHistories-v1` checks if the state plan is consistent (line 8). If so, it continues to find a root linear extension (lines 9-14). If not, it queues nothing into the stack, which forces the first invocation of `UnifyHistories-v1` to say there is no consistent goal-episode ordering.

The first linear extension can be found by exploiting the well known fact that the distance computed by SSSP, from one start-vertex to all other vertices, is a temporally consistent schedule for the events. `initUnifyHistories-v1` uses the distance values d cached as a part of ITC to find a temporally consistent schedule. The schedule admits an ordering over the goal-episode's start and end events. That event ordering can then be traversed by `ProjectStateConstraints` (Algorithm 7) to test for state-constraint consistency. The result are projected state-constraints that are then used

by `ComputeOrderingEpisodes` (Algorithm 8) to compute the ordering episodes that need to be added to the state plan.

`UnifyHistories-v1` (Algorithm 9), which incrementally computes the next consistent goal-episode ordering proceeds very similarly to `nextITO`, with the exception that it uses `UpdateOrderingFromSwap` (Algorithm 10) and `UpdateOrderingFromInsert` (Algorithm 11) to efficiently manage the projected state constraints and ordering episodes.

Note that earlier in this chapter we mentioned that we would use a SAT solver to help check for state-constraint consistency. That solver is used in the if-statements of `ComputeOrderingEpisodes` (Algorithm 7) line 3 and is implicitly used in `UpdateOrderingFromSwap` (Algorithm 8) and `UpdateOrderingFromInsert` (Algorithm 11) each time the array of projected state constraints is altered.

While `UnifyHistories-v1` benefits from the efficiency of incremental linear extension generation, it does not benefit from knowledge that can be extracted from temporal consistency checking (via ITC) or state-constraint consistency checking (via satisfiability). From ITC, we can learn sets of event orderings that should be disallowed because they cause temporal inconsistencies. From resolving the state-constraints, we can learn which episodes should not overlap and therefore which sets of events should not be interleaved. In UnifyHistories-v2 we present our approach to incorporating this information by modifying the way we test for partial-order relations (Algorithm 9, line 13).

4.6 UnifyHistories-v2

One of the benefits of building our `UnifyHistories-v1` algorithm based on `FindAllLinearExtensions` is that we can exploit the systematic nature in which total orders are generated. Despite all of the modifications we have made to that

Algorithm 6: initUnifyHistories-v1

Input: SP, EP_G
Output: void
// create a poset $\mathcal{P} = (S, R)$
1 $S = EV_G = \bigcup_{ep \in EP_G} \{ep.e_s, ep.e_e\}$;
2 $R = \emptyset$;
3 **for** ***each*** $ep = \{e_s, e_e, lb, ub, sc\}$ ***in*** EP_G **do**
4 **if** $lb > 0$ **then**
5 $R = R \cup (e_s, e_e)$
6 **else if** $lb < 0$ **then**
7 $R = R \cup (e_e, e_s)$
// find an initial total order (linear extension)
// by checking the temporal consistency of SP
8 **if** *CheckTemporalConsistency()* **then**
9 L_r = sort $EV_G = \{e_1, e_2, ..., e_n\}$ by increasing $d(e_i)$;
10 SC = ProjectStateConstraints(L_r,EP_G);
11 EP_{order} = ComputeOrderingEpisodes(L_r, SC);
12 $\mathcal{S} = \emptyset$;
13 $\mathcal{S}$.push($\langle L_r, n, 0, n \rangle$);
 // create the persistent data structure for Unify state
14 $ST = \langle R, \mathcal{S}, firstrun? = true, EP_G, SC, EP_{order} \rangle$;
15 **else**
 // create the persistent data structure for Unify state
 // with an empty stack
16 $ST = \langle R, \mathcal{S} = \emptyset, firstrun? = true, EP_G, SC[0], EP_{order} = \emptyset \rangle$;
// return the state plan with Unify state
17 $SP.Unifystate = ST$;
18 **return** SP

Algorithm 7: ProjectStateConstraints

Input: $L = (e_1, e_2, ..., e_n)$,$EP_G = \{ep_1, ep_2, ..., ep_m\}$
Output: SC, an array to cache the n-1 projected state-constraints.

```
1  EP_started = ∅;
2  SC_started = true;
3  SC=array[n-1];
4  i = 1;
5  for each e in L do
       // Check for an episode with start event e
6      for all ep : (ep ∈ EP_G ∧ e is ep.e_s) do
7          EP_started = EP_started ∪ ep;
8          SC_started = SC_started ∧ ep.sc
       // Check for an episode with end event e
9      for all ep : (ep ∈ EP_started ∧ e is ep.e_e) do
10         EP_started = EP_started \ ep;
11         SC_started = SC_started \ ep.sc
12     if i < n then
13         SC[i] = SC_started;
14         i = i + 1;
15 return SC
```

original algorithm, the total orders of goal events are still generated in the same systematic manner.

The conflict directed `UnifyHistories-v2` requires only a small modification to `UnifyHistories-v1`. In `UnifyHistories-v1` (Algorithm 9), line 13 is responsible for pruning total orders from being generated. The line checks whether "$j < n$ and $(e_j, e_{j+1}) \notin R$" is true. If true, events e_j and e_{j+1} are swaped to create the next total order. If false, a portion of the tree of total orders is automatically pruned and subsequent total orders are generated. In order to make Unified Histories conflict-directed, we can modify this check by adding an additional function `CanSwap?`. The new check for line 13 is "$j < n \wedge (e_j, e_{j+1}) \notin R \wedge$ `CanSwap?`(e_j, e_{j+1}, L)".

The `CanSwap?` algorithm allows us to declare whether the swapping of e_j and e_{j+1}

Algorithm 8: ComputeOrderingEpisodes

Input: $L = (e_1, e_2, ..., e_n), SC = [sc_1, sc_2, ...sc_{n-1}]$
Output: EP_{order}

```
1 EP_order = ∅ ;
2 for each i in {1, 2, ..., n − 1} do
3     if sc_i == false then
4         ep_sequential = ⟨e_i, e_{i+1}, 0, 0, true⟩;
5         EP_order = EP_order ∪ {ep_sequential};
6     else
7         ep_co-occur = ⟨e_i, e_{i+1}, 0, ∞, true⟩;
8         EP_order = EP_order ∪ {ep_co-occur};
9 return EP_order;
```

should be permitted relative to the current state of the total order, L.

To maximize the generality of `CanSwap?`, we introduce the notion of an *ordering conflict function*, $oc : EV \times EV \times 2^L \rightarrow \{true, false\}$, which has the same inputs, outputs, and behavior as `CanSwap?` but allows us to allocate one function per conflict to ensure the conflict is avoided in future total orders. These ordering conflicts are stored in a list, OC. `CanSwap?` computes the conjunction of the function in this list (Algorithm 12).

One important conflict for `UnifyHistories-v2` to avoid is overlapping episodes when their state-constraints are trivially unachievable. We can now enforce the idea that episodes should not overlap by introducing an ordering conflict function tailored to this conflict.

Let's take for example a pair of episodes $ep_{12} = \langle e_1, e_2, l_{12}, u_{12}, sc_{12} \rangle$ and $ep_{34} = \langle e_3, e_4, l_{34}, u_{34}, sc_{34} \rangle$ where $sc_{12} \wedge sc_{34} = false$, $l_{12} < u_{12}$, and $l_{34} < u_{34}$. In other words, the state constraints are trivially unachievable and the start events must temporally preceed the end events. The only two consistent orderings possible for these episodes are (e_1, e_2, e_3, e_4) and (e_3, e_4, e_1, e_2).

Figure 4-10 depicts the tree of total orders that will be traversed to find these

Algorithm 9: UnifyHistories-v1

Input: SP

Output: SP' or `null` if no next total order exists

$\langle R, \mathcal{S}, firstrun?, EP_G, SC, EP_{order} \rangle = ST = SP.ITOstate$;
if $\mathcal{S}.isempty()$ **then**
    return *null*;
$\langle L, l_p, i, j \rangle \leftarrow \mathcal{S}$.pop(); // L=$(e_1, e_2, ..., e_n)$
if *firstrun?* **then**
    $firstrun? = false$;
    $\mathcal{S}$.push($\langle L, l_p, i, j \rangle$);
    // Induce total order over goal episodes by adding ordering episodes. Let $SP = \langle GH, \mathcal{VH}, \mathcal{J} \rangle$ and $GH = \langle EV_{GH}, EP_{GH} \rangle$
    $SP.GH.EP_{GH} = SP.GH.EP_{GH} \cup EP_{order}$;
    if CheckTemporalConsistency(SP) **then**
        return SP;
    else
        return UnifyHistories-v1(SP);
if $j < n$ *and* $(e_j, e_{j+1}) \notin R$ **then**
    swap e_j and e_{j+1} in L;
    $\langle SC, EP_{order} \rangle$ = UpdateOrderingFromSwap($SC, EP_{order}, L, EP_G, j, j+1$);
    j=j+1;
    $\mathcal{S}$.push($\langle L, l_p, i, j \rangle$);
    $\mathcal{S}$.push($\langle L, i, 0, n \rangle$);
    if CheckTemporalConsistency(SP) **then**
        return SP;
    else
        return UnifyHistories-v1(SP);
else
    if $i > 0$ **then**
        insert e_j immediately after e_{i-1} in L;
        // now $e_i = i$ again holds, and L is restored as it was
        $\langle SC, EP_{order} \rangle$ = UpdateOrderingFromInsert($SC, EP_{order}, L, EP_G, j, i-1$);
    i=i+1;
    if $i < l_p$ **then**
        j=i;
        $\mathcal{S}$.push($\langle L, l_p, i, j \rangle$);
    return UnifyHistories-v1(SP);

Algorithm 10: UpdateOrderingFromSwap

Input: $SC = [sc_1, ...sc_{n-1}], EP_{order}, L = (e_1, ..., e_n), EP_G = \{ep_1, ..., ep_m\}, j, j'$

// $(e_{j'}, e_j)$ was the previous event ordering
// $(e_j, e_{j'})$ is the current ordering in L, after the swap.
// first, update cache of state constraints
// $e_{j'}$ starts an episode and has moved 'to the right' in L
1 **for** ***all*** $ep : (ep \in EP_G \wedge e_{j'}$ *is* $ep.e_s)$ **do**
2 $\quad SC[j] = SC[j] \setminus ep.sc;$
// $e_{j'}$ ends an episode and has moved 'to the right' in L
3 **for** ***all*** $ep : (ep \in EP_G \wedge e_{j'}$ *is* $ep.e_e)$ **do**
4 $\quad SC[j] = SC[j] \cup ep.sc;$
// e_j starts an episode and has moved 'to the left' in L
5 **for** ***all*** $ep : (ep \in EP_G \wedge e_j$ *is* $ep.e_s)$ **do**
6 $\quad SC[j] = SC[j] \cup ep.sc;$
// e_j ends an episode and has moved 'to the left' in L
7 **for** ***all*** $ep : (ep \in EP_G \wedge e_j$ *is* $ep.e_e)$ **do**
8 $\quad SC[j] = SC[j] \setminus ep.sc;$
// second, update ordering episodes
// update previous orders
9 **if** $j - 1 >= 1$ **then**
10 $\quad$ Find the episode ep in EP_{order} that starts at e_{j-1} and ends at $e_{j'}$.
11 $\quad ep.e_e = e_j;$
12 **if** $j + 1 <= n$ **then**
13 $\quad$ Find the episode ep in EP_{order} that starts at e_j and ends at $e_{j'+1}$.
14 $\quad ep.e_s = e'_j;$
// replace the old order with a new order, representing the swap
15 Find the episode ep in EP_{order} that starts at $e_{j'}$ and ends at e_j.
16 $EP_{order} = EP_{order} \setminus ep;$
17 **if** $SC[j] \neq$ ***false*** **then**
18 $\quad ep_{sequential} = \langle e_i, e_{i+1}, 0, 0, \textbf{true} \rangle;$
19 $\quad EP_{order} = EP_{order} \cup \{ep_{sequential}\};$
20 **else**
21 $\quad ep_{co\text{-}occur} = \langle e_i, e_{i+1}, 0, \infty, \textbf{true} \rangle;$
22 $\quad EP_{order} = EP_{order} \cup \{ep_{co\text{-}occur}\};$
23 **return** $\langle SC, EP_{order} \rangle;$

Algorithm 11: UpdateOrderingFromInsert

Input: $SC = [sc_1, ...sc_{n-1}], EP_{order}, L = (e_1, ..., e_n), EP_G = \{ep_1, ..., ep_m\}, l, k$

// $(..., e_k, e_{k+1}, e_{k+2}, ..., e_l, ...)$ is the current ordering in L, after the insert.

// $(..., e_k, e_{k+2}, ..., e_{l-1}, e_{k+1}...)$ was the previous event ordering

// first, update cache of state constraints

// shift a subsequence of SC 'to the right'

$SC[k+2...l-1] = SC[k+1..l-2]$;

// e_l starts an episode and has moved 'to the left' in L

for ***all*** $ep : (ep \in EP_G \wedge e_l$ *is* $ep.e_s)$ **do**
    for i ***from*** *k+2* ***to*** *l-1* **do**
        $SC[i] = SC[i] \setminus ep.sc$;

// e_l ends an episode and has moved 'to the left' in L

for ***all*** $ep : (ep \in EP_G \wedge e_l$ *is* $ep.e_e)$ **do**
    for i ***from*** *k+2* ***to*** *l-1* **do**
        $SC[i] = SC[i] \setminus ep.sc$;

// second, update ordering episodes

// remove previous orderings

Find the episode ep in EP_{order} that starts at e_{l-1} and ends at e_l.

$EP_{order} = EP_{order} \setminus ep$;

Find the episode ep in EP_{order} that starts at e_k and ends at e_{k+2}.

$EP_{order} = EP_{order} \setminus ep$

// add new orderings

if $SC[k] \neq$ ***false*** **then**
    $ep_{sequential} = \langle e_k, e_{k+1}, 0, 0, \textbf{true} \rangle$;
    $EP_{order} = EP_{order} \cup \{ep_{sequential}\}$;
else
    $ep_{co\text{-}occur} = \langle e_k, e_{k+1}, 0, \infty, \textbf{true} \rangle$;
    $EP_{order} = EP_{order} \cup \{ep_{co\text{-}occur}\}$;

if $SC[k+1] \neq$ ***false*** **then**
    $ep_{sequential} = \langle e_{k+1}, e_{k+2}, 0, 0, \textbf{true} \rangle$;
    $EP_{order} = EP_{order} \cup \{ep_{sequential}\}$;
else
    $ep_{co\text{-}occur} = \langle e_{k+1}, e_{k+2}, 0, \infty, \textbf{true} \rangle$;
    $EP_{order} = EP_{order} \cup \{ep_{co\text{-}occur}\}$;

return $\langle SC, EP_{order} \rangle$;

Algorithm 12: CanSwap?

Input: $e_i \in EV_G$
Input: $e_j \in EV_G$
Input: L, the 'current' total order of EV_G
1 **return** $\bigwedge_{oc \in OC} oc(e_i, e_j, L)$

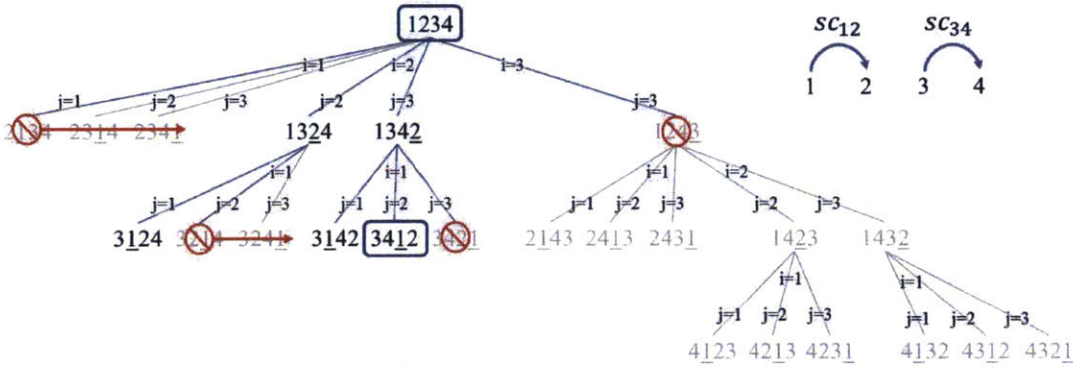


Figure 4-10: The tree of total orders for the two goal episodes (upper right) which cannot be achieved at the same time. The blue rectangles denote the orderings we want to generate. The red circles denote orders already pruned by the temporal constraints on the episodes. The remaining orderings must be pruned by an ordering conflict function.

two orders. The blue circles denote the orderings we want to generate. The red circles denote orders already pruned by the temporal constraints on the episodes. The remaining orderings must be pruned by our ordering conflict function.

Algorithm 13 implements the conflict ordering function which prevents these two episodes from temporally overlapping. It allows any swaps that preserve the non-overlapping ordering.

Algorithm 13: SCOrderingConflict

// ep_{12} and ep_{34} are two goal episodes that should not overlap.
Data: $ep_{12} = \langle e_1, e_2, l_{12}, u_{12}, sc_{12} \rangle$
Data: $ep_{34} = \langle e_3, e_4, l_{34}, u_{34}, sc_{34} \rangle$
Input: $e_i \in EV_G$
Input: $e_j \in EV_G$
Input: L, the 'current' total order of EV_G

```
1  i_1 = current index of e_1 in L;
2  i_2 = current index of e_2 in L;
3  i_3 = current index of e_3 in L;
4  i_4 = current index of e_4 in L;
5  if i_1 < i_2 < i_3 < i_4 then
6  |  if e_i = e_2 ∧ e_j = e_3 then
7  |  |  return false
8  |  else if e_i = e_3 ∧ e_j = e_2 then
9  |  |  return true
10 |  else return true
11 else if i_3 < i_4 < i_1 < i_2 then
12 |  if e_i = e_4 ∧ e_j = e_1 then
13 |  |  return false
14 |  else if e_i = e_1 ∧ e_j = e_4 then
15 |  |  return true
16 |  else return true
17 else return false
```

Chapter 5

Incremental Temporal Consistency v2: Efficiently Checking the Temporal Consistency of a Plan

5.1 Introduction

An important component in the tBurton algorithm is the ability to check whether a plan is temporally consistent. During the planning process, tBurton's high-level search attempts to find sequences of state transitions that achieve both particular states, but also achieves them at particular times. Recall, in our vernacular, we refer to this process as *closing* a *goal-episode*, where a goal-episode represents both the desired state through its state constraint, and the desired duration through its set-bounded temporal constraint. While the bulk of tBurton's high-level search focuses its effort on creating a plan that satisfies the state-constraints, it relies on Incremental Temporal Consistency v2 (ITCv2) to check whether the plan also satisfies the temporal constraints.

Given a plan, it's the job of ITCv2 to say whether that plan is temporally consistent. For example, a plan that requires five hours to prepare and cook a turkey and also requires that turkey be ready for guests in at most two hours, is not temporally consistent.

More generally, we say ITCv2 checks a Simple Temporal Network (STN), the temporal constraints that underly the plan, and returns two values. The first value indicates whether the plan is temporally consistent (true if consistent, false otherwise). If the plan is inconsistent, the second value identifies one subset of the temporal constraints responsible for that inconsistency. Knowledge of why an inconsistency exists can be very effective at guiding search [52, 46, 60].

While determining whether a STN is temporally consistent from scratch is useful, sometimes only minor changes will be made to the STN before checking again for consistency. Such minor changes often occur in search. In these cases, an *incremental* temporal consistency checker that could be notified of those changes, and leverage the work from previous invocations, could save a lot of time. For tBurton, which can add/remove episodes (temporal constraints) to very large partial plans, an incremental temporal consistency checker is essential. The 'I' in ITCv2 stands for Incremental.

In this chapter we present the ITCv2 algorithm. We start by providing some informal background on how consistency checking works and then use this background to make the innovations of ITCv2 distinct from its predecessor, ITC [52]. We then discuss some related work to clarify our terminology before introducing some formal notation. The bulk of the chapter presents the ITCv2 algorithm. Finally, we close with some empirical results.

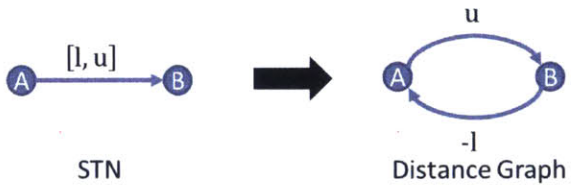


Figure 5-1: The basic building block of the translation from STN to Distance Graph

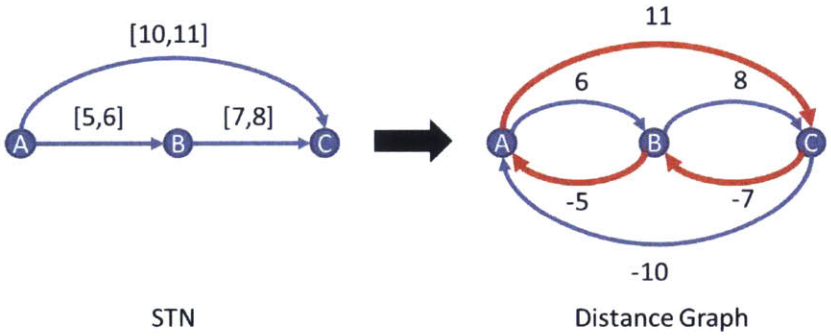


Figure 5-2: An example of checking for the consistency of an STN. This STN is inconsistent, as indicated by the bold, red edges corresponding to one infeasible subset of the temporal constraints.

5.2 Background and Motivation

5.2.1 Temporal Consistency with Negative Cycle Detection

The basis for checking whether a Simple Temporal Network (STN) is temporally consistent was described and rigorously proven by Dechter, Meiri, and Pearl [17]. The process involves translating a STN into an equivalent representation called a distance graph (a weighted directed graph) and then checking that graph for negative cycles. The existence of a negative cycle means the STN is temporally inconsistent. For brevity, we will drop the word *temporally*, and simply say a STN is consistent/inconsistent.

Figure 5-1 depicts the basic building block of the translation from a STN to a

distance graph. Informally, a STN is a set of time-points related by a set of temporal constraints that dictate the relative duration between pairs of time points. The left side of figure 5-1 depicts a very simple STN as a graph with two vertices A and B representing time-points, and an edge $A \xrightarrow{[l,u]} B$ representing the temporal constraint $l \leq B - A \leq u$. The right side of figure 5-1 depicts the distance graph for the STN. The distance graph uses the same vertices as the STN, but contains a pair of edges: $A \xrightarrow{u} B$ represents the $B - A \leq u$ relation of the temporal constraint and $B \xrightarrow{-l} A$ represents $A - B \leq -l$. By re-interpreting the edge-labels in the distance graph as distances instead of temporal constraints, we can now ask the question whether there is a negative cycle in the distance graph. If it is possible to traverse the distance graph in some way as to start and end at the same vertex, yet have walked a negative distance, we can conclude the original STN is inconsistent. Figure 5-2 depicts a more complex temporal network, its equivalent distance graph, and a negative cycle. The negative cycle identifies an inconsistency in the STN, where the set of temporal constraints requires time-point C occur both at most 11 after A and at least 12 after A.

While it is straight-forward to visually inspect and find negative cycles in a small distance graph, this quickly becomes impossible as the size of the graph grows. The difficulty in designing an incremental temporal consistency checker therefore lies in the design of the negative cycle detection algorithm.

5.2.2 ITC

The ITCv2 algorithm we describe in this chapter is an evolution of Incremental Temporal Consistency (ITC), an algorithm developed by Shu and Williams [52]. The original ITC [17] was based on a combination of Bellman-Ford-Moore's (BFM) Single Source Shortest Path (SSSP) algorithm and a simple negative cycle detection strat-

egy. Given a starting vertex, ITC would use BFM to find the shortest path from the starting vertex to all other vertices. The BFM algorithm, also called a label-correcting algorithm, associates a number with each vertex indicating the shortest path-length yet discovered to that vertex. During a run of BFM, a negative cycle can be detected if any label value falls below -nC [1], where n is the number of vertices in the distance graph (time-points in the STN) and C is the largest positive edge weight (largest upper bound in the STN). For a distance graph with n vertices and m edges (and without negative cycles), the runtime complexity of BFM is O(nm). [2]

To make the algorithm incremental, ITC leverages the labels as well as the queue BFM uses to keep track of vertices to explore. If an edge is added to the distance graph that creates a shorter path than already discovered, vertices are enqueued so BFM may correct its SSSP computation and update affected labels. If an edge is removed from the distance graph which was involved in a shortest path, affected labels are reset to infinity, and some vertices are again enqueued so BFM may update its labels. Finally, if a negative cycle is discovered, the parent vertices of vertices in the negative cycle are enqueued, and the labels of all vertices that could be affected by the negative cycle are reset to infinity before invoking BFM.

The key principle behind the design of ITC's incremental operation is that BFM monotonically decrements the labels until they accurately represent the shortest path distance from the start vertex to any other vertex. Thus any change that made a subset of the labels inaccurate required two fixes: 1. Depending on the change, specific vertices needed to be re-queued so BFM could discover which labels to fix. 2. Any labels that could be lower than their correct shortest-path value needed to be

[1]The theoretical bound for this negative cycle detection strategy is actually $-(n-1)C$, but we will use -nC, as reported in the original ITC paper.

[2]The complexity of BFM is sometimes also reported as $O(n^3)$, where it is assumed that in the worse case, a fully-connected graph has $m = n^2$ edges. However, BFM's runtime is much more accurately described by $O(nm)$, as there are other algorithms such as Floyd-Warshall which are inherently $O(n^3)$

reset to infinity because BFM can only converge on the correct label value through decrement.

To compute a set of inconsistent constraints from a distance graph that contains a negative cycle, ITC must identify the edges involved in the negative cycle. This is easily done by having BFM maintain a shortest-path tree (sometimes also referred to as back-pointers or parent-pointers). Once a negative cycle is detected, ITC traverses the shortest-path tree starting at the vertex with label less than -nC until the same vertex is reached. Any edge walked in the traversal is involved in the negative cycle and represents a temporal inconsistency.

5.2.3 ITC to ITCv2

ITCv2 inherits many core ideas from ITC, but improves upon ITC in several ways.

Functionally, ITCv2 and ITC are very similar: They have the same input/output behavior; Both use Bellman-Ford-Moore (BFM) as the underlying SSSP label-correction algorithm; and both use the same incremental strategy of selectively enqueing vertices for label-correction by BFM after a change.

ITCv2 differs from its predecessor by improving on four shortcoming in ITC's algorithm.

The first, and simplest to solve, shortcoming of the original ITC is that its dependence on a start-vertex can miss detecting negative cycles in portions of the distance graph unreachable from the start vertex. A well known solution to this problem is to introduce a dummy start vertex with an outgoing edge of weight 0 to all vertices in the original distance graph. We take this approach one step further and simply enqueue all the vertices in the distance graph with labels of 0.

The second shortcoming of ITC is that there are better known variants of Bellman-Ford-Moore. ITCv2 takes advantage of Goldberg-Radzik's algorithm [28], a variation

on BFM which uses two queues in a heuristic based approach that attempts to expand each vertex only once. Since a distance graph derived from an STN is usually well connected because of the pair of edges corresponding to both the lower and upper bounds of a temporal constraint, a vertex can be expanded several times before its label converges on the shortest-path value. Reducing the number of vertex expansions can have a significant impact on performance. Empirical analysis reflects the gains that Goldberg-Radzik can have over plain BFM [10].

The third shortcoming of the original ITC is its use of -nC negative cycle detection. In the event of a negative cycle, BFM can easily take much greater than O(nm) time, needlessly cycling. We can prevent this (as acknowledged in the original ITC paper [52]) by maintaining a shortest-path tree (rooted at the dummy start vertex). If the shortest-path tree ever connects back on itself, we know a negative cycle exists.

The fourth shortcoming of the original ITC is that once a negative cycle is detected, many vertices are queued and many labels are reset to ∞. Recall that resetting a label to ∞ was necessary because BFM (and Goldberg-Radzik) will only decrement label values. The concern in the original ITC was that before BFM concluded a negative cycle existed, BFM could reduce the label values of many verticies to a value below their correct shortest-path value (if the negative cycle did not exist). In ITCv2, we use Tarjan's Subtree Disassembly [53] to detect negative cycles early. Any vertex labels impacted by constraints in the negative cycle are naturally taken care of by the procedures needed to incrementally add/remove an edge. Subtree Disassembly has the added benefit of complimenting Goldberg-Radzik's.

5.3 Related Work

5.3.1 Full Dynamic Consistency

The problem of incremental temporal consistency we have presented is also commonly referred to as a *fully dynamic* consistency-checking problem. Fully-dynamic consistency-checking has been studied for a variety of problems including APSP, SSSP, as well as difference-constraint solvers [23, 37, 54].

In these works, the term fully dynamic is used to refer to an algorithm that can respond to *incremental* and *decremental* change. The term incremental now refers to inserting a new edge or decreasing the weight of an existing edge (tightening a set of constraints), and decremental refers to removing an edge or increasing the weight of an existing edge (loosening a set of constraints).

Relative to this terminology, ITCv2 is fully dynamic, and can further tolerate multiple incremental and decremental changes prior to re-establishing consistency. We will use the terms add or tighten a constraint in place of incremental, and remove or loosen a constraint in place of decremental.

While we build ITCv2 based on the foundations of ITC, tt is worth mentioning that fully dynamic ITC (although not referred to as such) has been addressed before by Ramalingam et. al. in 1999 [48]. Unlike ITC, which bases its graph traversal and negative cycle detection strategy on BFM, they base their algorithm on Dijkstra's SSSP algorithm. Dijkstra's algorithm has a more favorable $O(m + nlogn)$ runtime, compared to BFM's $O(nm)$ runtime, but is not usually prescribed for graphs with negative edge weights. Their work avoids this restriction by requiring the initial distance-graph they must check be consistent, allowing them to detect future inconsistencies as a result of change before running Dijkstra's. The assumption that the initial distance-graph or simple temporal network is consistent is quite reasonable.

Relative to tBurton, our initial temporal network comes from the user in the form of a partial state plan. This state plan could be checked for consistency (i.e. using BFM) prior to using their faster approach. However, their approach does not identify and return the cause of the inconsistency, a negative cycle.

5.3.2 Negative Cycle Detection

A variety of algorithms for detecting negative cycles have been presented in the path planning and graph-theory literature. These algorithms all combine a shortest path algorithm with a cycle detection strategy [10]. Despite being a combination, the complexity of negative cycle detection is usually dominated by the shortest-path algorithm which exerts more effort in traversing the graph.

The choice to use Goldberg-Radzik's algorithm and Tarjan's subtree disassembly for ITCv2 required consideration of not only theoretical complexity, but practical performance on graphs with common cycle structures as those found in STN derived distance graphs.

Floyd-Warshall's all-pairs shortest-path (APSP) [20] algorithm computes a matrix that associates with each pair of verticies, the shortest-path distance between them. A negative entry in the diagonal of this matrix indiciates a negative cycle exists. In general, for graphs with no restrictions on the range of edge weights or connectivity, Floyd-Warshall has $O(n^3)$ runtime. Alternative APSP algorithms, such as Johnson's APSP algorithm, has a better runtime, $O(n^2 log n + nm)$. To our knowledge, the fastest known runtime for APSP comes from work in fully dynamic APSP, which achieves an amortized $O(n^{2.75})$ [54]. Regardless the runtime, the fact of the matter is APSP computes more information than is needed for negative cycle detection.

The best known theoretical time-bound for negative cycle detection comes from Bellman-Ford-Moore, with $O(nm)$ runtime. Specific variations of BFM, such Gold-

berg's algorithm can improve these bounds to $O(\sqrt{n}m \log N)$ if the edge weights are integers and bounded by $-N \leq -2$ [27]. Further average-case runtime analysis based on input distributions of real-valued edge weights has been studied that further differentiates variations of BFM [38]. The algorthm we use, Goldberg-Radzik, has the same worse-case bounds as BFM, but incorporates a heuristic that attempts to only expand each vertex once.

To complement the diversity of shortest-path algorithms, there is also a diversity of cycle detection strategies. We have already described -nC. Another approach is to walk to the root: before a vertex is expanded, traverse the parent-pointers maintained by BFM until the root is reached or a cycle is detected. Alternatively, one can maintain child-pointers (the reverse of parent-pointers), and traverse the shortest-path subtree before each vertex is expanded to discover cycles. This alternative approach is better than walking to the root if the cycles are local to the vertex. Tarjan's subtree disassembly is a variation on walking the shortest-path subtree. As the subtree of a given vertex is walked and checked for a negative cycle, all vertices in the subtree are also removed from BFM's queue. This prevents the early expansion of vertices. There are yet other approaches, and approaches that are variations on these which attempt to amortize the cost of negative cycle detection, or incorporate that cost into the operation of BFM.

To make sense of this diversity, and the combinations of shortest path and negative cycle detection algorithms, we examined work that surveyed these combinations and tested them empirically [10, 40]. Goldberg-Radzik with Tarjan's subtree disassembly consistently outperforms other approaches for a variety of graphs, but also performs well for graphs with many negative and positive cycles.

5.4 Notation and Definitions

Now that we've sketched out the functionality of ITC and its ITCv2 extension, we return to formalize those ideas. We start by establishing the graph notation we will use, in the context of the inputs and outputs of ITCv2. We then provide the standard definitions of an STN and its distance graph representation. Finally, we end this subsection by formalizing the mapping of a state-plan to an STN, completing the story of how temporal consistency checking can be applied to a partial plan.

5.4.1 Problem Formulation for ITCv2

The input to ITCv2 is $\langle G, w \rangle$, where $G = \langle V, E \rangle$ is a directed graph and $w : E \rightarrow \mathbb{R}$ is the weight function. An edge $(x, y) \in E$ is a defined by a pair of vertices, $x, y \in V$. We will use $pred : V \rightarrow 2^V$, the predecessor function, to denote the set of vertices $\{x | (x, y) \in E\}$ that are predecessors of vertex y.

Traditionally, the input to BFM, on which ITCv2 is based, also includes the start vertex, s_0. We leave it out of our input definition because ITCv2 will always use a dummy start-vertex. For now, we will use s_0 to denote this dummy start vertex.

The output from ITCv2 is a negative cycle, $negcycle = ((v_1, v_2), (v_2, v_3), ..., (v_k, v_1))$, $v_i \in V$, a 'cyclic' sequence of all edges participating in the negative cycle. If there is no negative cycle, $\emptyset$ is returned. Since a negative cycle is just a special case of a *path*, this definition reflects the definition of a path as a sequence of edges, with the additional requirement that the last edge completes the cycle by sharing a vertex with the first edge.

ITCv2 will also store several values related to the graph. $d : V \rightarrow \mathbb{R}$ is the labeling or distance function, which represents the distance of a vertex from s_0. $p : V \rightarrow V \cup unknown$, is the parent function. For a vertex y, $p(y)$ returns a vertex $x \in pred(y)$ that occurs along the current shortest known path from s_0 to y. $c : V \rightarrow 2^V$, is the

children function. The children of x are $\{y|x = p(y)\}$, the set of vertices that have x as their parent.

While the parent function is traditionally used in shortest path algorithms to recover the shortest path, the children function, which stores 'pointers in the opposite direction', will make techniques like Tarjan's subtree disassembly possible. Both the parent and the children function will be needed to recover the negative cycle from ITCv2.

5.4.2 Reducing State Plans to Distance Graphs

Until now we have abstracted away the issue of calling ITCv2 on State Plans. We return to formalize this mapping. We first define an STN and its corresponding Distance Graph. We then define the mapping from a State Plan to an STN.

Definition 21. (Simple Temporal Network) A Simple Temporal Network (STN) is a tuple $\langle \mathcal{T}, \mathcal{C} \rangle$, where $\mathcal{T}$ is a set of real-valued variables called time-points and $\mathcal{C}$ is set of *temporal constraints*, each of the form $l \leq T_j - T_i \leq u$, where $T_i, T_j \in \mathcal{T}$ and $l, u \in \mathbb{R}$. We refer to l and u as the *lower bounds* and *upper bounds*, respectively, on the duration that can separate the two events.

Definition 22. (Distance Graph) The Distance Graph of an STN $\langle \mathcal{T}, \mathcal{C} \rangle$ is the tuple $\langle DG, w \rangle$, where $DG = \langle \mathcal{T}, \mathcal{E} \rangle$ is a directed graph consisting of a set vertices $\mathcal{T}$ and a set of edges $\mathcal{E}$. (T_i, T_j) and (T_j, T_i) are elements of $\mathcal{E}$ with $w(T_i, T_j) = u$ and $w(T_j, T_i) = -l$ iff $l \leq T_j - T_i \leq u$ is a temporal constraint in $\mathcal{C}$.

Recall that a State Plan is the tuple $\langle GH, \mathcal{VH}, \mathcal{J} \rangle$, where $GH = \langle EV_{GH}, EP_{GH} \rangle$ is a goal history, $VH \in \mathcal{VH}$ is a set of value histories of the form $VH = \langle EV_{VH}, EP_{VH} \rangle$, and $EP_J \in \mathcal{J}$ is a set of justifications which take the form of episodes (Definition 4). An episode, $ep \in EP$ takes the form $ep = \langle e_i, e_j, lb, ub, sc \rangle$.

An STN $\langle \mathcal{T}, \mathcal{C} \rangle$ can be extracted given all of the episodes from the goal history, value histories, and justifications. Specifically, $e_i, e_j \in \mathcal{T}$ and $[lb \leq e_j - e_i \leq ub] \in \mathcal{C}$ iff an episode $ep = \langle e_i, e_j, lb, ub, sc \rangle$ participates in the State Plan.

5.5 ITCv2 Algorithm

5.5.1 Solving the Shortcomings of the Original ITC

Reconnecting Disconnected Distance Graphs

The first, and simplest to solve shortcoming of the original ITC is that it does not work on disconnected distance graphs. To be more precise, only negative cycles reachable from the start-vertex can be detected. Figure 5-3 shows a common example of how such a distance graph can be created. An STN with a temporal constraint where the upper bound is ∞ is equivalent to not having an edge in the distance graph. In this example, a poor selection of vertex A as the start-vertex has left the C-D subgraph unreachable by BFM. While it is correct for a SSSP algorithm to declare that vertices C and D are unreachable from A, the failure of BFM to traverse the C-D subgraph means it also misses detecting the negative cycle. A well known solution to this problem is to introduce a dummy start vertex with an outgoing edge of weight 0 to all vertices in the original distance graph.

This approach works for two reasons. First, all vertices are now reachable from the dummy start vertex and subject to negative cycle detection. Second, no new cycles are introduced. Since there is no edge returning to the dummy start vertex, there is no cycle that could include the new 0-weight edge.

We take this approach of using a dummy-start vertex one step further and simply initialize ITC-v2 by enqueing all the vertices in the distance graph with labels of 0. This simulates the effect of one pass of BFM without having to introduce a dummy

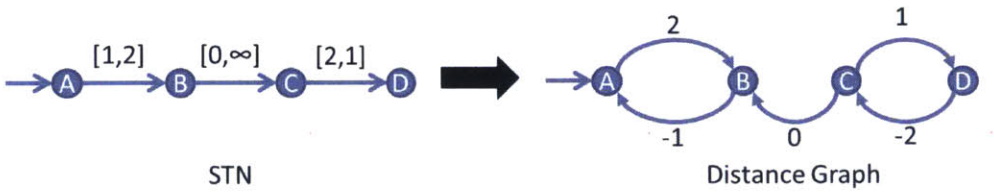


Figure 5-3: A temporal constraint with an unbounded upper bound can result in a distance graph with an unreachable subgraph.

start vertex and a bunch of 0-weight edges.

Understanding Goldberg-Radzik's Heuristic Queueing

To understand Goldberg-Radzik's algorithm, we start with a brief background on BFM. BFM is considered a scan-based shortest path algorithm. It iterates over a queue of vertices scanning each vertex one at a time. A scan consists of checking if a vertex u has a successor vertex v in which $d(v) > d(u) + w(u, v)$. If so, the distance label for v is updated $d(v) = d(u) + w(u, v)$, the parent of v is set $p(v) = u$, and v is queued. The checking and setting of $d(v)$ is called a *relaxation*, with v referred to as the relaxed vertex.

Goldberg-Radzik's algorithm is similarly scan-based, but uses two queues A and B. Initially $A = \emptyset$ and $B = \{s_0\}$. At the start of each scan, B is topologically sorted by depth from s_0, stored into A, and emptied. A therefore acts as the queue used in BFM. During a scan, the relaxed vertex is added to B if it is not already in B and not already queued for scanning in A. Once all vertices in A have been scanned, this process repeats until there are no more vertices in A to scan.

The idea behind Goldberg-Radzik is that relaxing vertices closer to s_0 will most likely result in the relaxation of vertices that are farther away. Instead of relaxing vertices farther away first, only to have to scan and relax them again when a vertex closer to s_0 is relaxed, make sure vertices closer to s_0 are relaxed first. This heuristic

is implemented through the topological sort of queue B.

Incorporating Tarjan's Subtree Disassembly

The ideal SSSP algorithm would only have to relax each vertex once. Unfortunately, no one knows how to accomplish this feat. As a consolation prize, variants of BFM strive to minimize the number of times each vertex needs to be relaxed. Tarjan's subtree disassembly is an addition to BFM that attempts to minimize the number of times each vertex needs to be relaxed. It does this by taking advantage of the shortest path tree already being built by BFM. When a vertex v is relaxed, all vertices in its shortest path tree rooted at v are removed from the queues. The idea is: When v is relaxed, all vertices in the subtree rooted at v must also need their distance labeled reduced. Since relaxing v entails queuing v for the next scan, reducing any vertices from v's subtree in the current scan is a waste of time. They must be revisited in subsequent scans, but perhaps they will be relaxed fewer times.

Tarjan's subtree disassembly also doubles as a great negative cycle detector. Imagine we are scanning vertex u and have decided that its successor vertex v needs to be relaxed. If u is also a vertex in the subtree rooted at v, then we have identified a negative cycle. The edges involved in the cycle are the edges from v to u in the shortest path tree concatenated with the edge (u, v). For subsequent exposition, we will refer to this (u, v) edge as the 'last' edge needed to close the cycle.

The proof is simple. Lets call the length of the path from u to v in the shortest path tree, l. Then $d(u) = d(v) + l$. But, the fact that we are about to relax v means $d(v) > d(u) + w(u, v)$. Combining these two equations, we get $l + w(u, v) < 0$, indicating the path length is negative. Since the path length is negative and the detection strategy looks for a cycle, we have detected a negative cycle. Note that even though the shortest path tree only contains the shortest path known at that

point, the length of any path in the tree can can only decrease as BFM runs. Since l will only decrease, a negative cycle detected by Tarjan's subtree disassembly will always be a negative cycle.

Concerning ITCv2, Tarjan's subtree disassembly fixes two problems from ITC. First, in terms of number of scans, it offers a significantly faster negative cycle detection strategy than $-nC$. Second, it obviates resetting edge weights to ∞. Tarjan's subtree disassembly allows the detection of negative cycles right before BFM relaxes the last edge to close the cycle. Since the negative cycle is never closed, the distance labels of vertices connected to the cycle would be the same as if that last edge did not exist. Therefore, it is unnecessary to reset any edge weights to ∞ when a negative cycle is detected.

5.5.2 Algorithm: ITCv2 with Goldberg-Radzik and Tarjan's Subtree Disassembly

ITCv2 consists of three important 'public-facing' algorithms. `Initialize` (Algorithm 14) initializes the data structures used by ITCv2 and implements our approach to fixing disconnected graphs. `CheckTemporalConsistency` (Algorithm 15) implements the bulk of the incremental algorithm to check for temporal consistency, including Goldberg-Radzik and Tarjan's subtree disassembly. `ModifyConstraint` (Algorithm 16) allows edges to be added, removed, and modified from the distance graph as if they were temporal constraints.

The remaining algorithms are used in support of these three public-facing algorithms. `ExtractNegativeCycle` (Algorithm 19) uses a mapping from parent to child verticies which caches the subtree from Tarjan's subtree disassembly to build a negative cycle. `DeleteSubtree` (Algorithm 20) traverses a subtree and ensures all child vertices are cleared from both A and B queues.

Algorithms 14-20 provides the pseudo code for ITCv2.

Algorithm 14: Initialize

Input: G
Output: void

```
1 A := ∅;
2 B := ∅;
3 for each s ∈ V(G) do
4 |   d(s) := 0;
5 |   p(s) := unknown;
6 |   c(s) := ∅;
7 |   B.enqueue(s);
```

Algorithm 15: CheckTemporalConsistency

```
   Input:
   Output: neg-cycle
 1 while B ≠ ∅ do
 2     A := topological-sort(B);
 3     B := ∅;
 4     while A ≠ ∅ do
 5         u := A.peek();
 6         for v ∈ succ(u) do
 7             if (d(u)+w(u,v))<d(v) then
 8                 pcmap := DeleteSubtree(v,∅);
 9                 if (u ∉ A) ∧ (u ∉ B) then
10                     B := B ∪ A;
11                     A.dequeue();
12                     B.enqueue(u);
13                     return ExtractNegativeCycle(pcmap,u,v);
14                 d(v) := d(u) + w(u,v);
15                 p(v) := u;
16                 c(u) := c(u) ∪ v;
17                 if (v ∉ A) ∧ (v ∉ B) then
18                     B.enqueue(v);
19         B := B\u;
20         A := A\u;
21 return ∅;
```

Algorithm 16: ModifyConstraint

```
   Input: x,y,l,u
   Output: void
 1 ModifyEdge(y,x,-l);
 2 ModifyEdge(x,y,u);
```

Algorithm 17: ModifyEdge

Input: x,y,w_{new}
Output: void

```
1  w(x,y) := w_new ;
2  if d(y) > d(x) + w_new then
3  |   d(y) := d(x) + w_new;
4  |   DeleteSubtree(y,∅);
5  |   p(y) := x;
6  |   c(x) := c(x) ∪ y;
7  |   B.enqueue(y);
8  else if (d(y) < d(x) + w_new) ∧ (p(y) == x) then
9  |   pcmap := DeleteSubtree(y,∅);
10 |   for all (p,c) ∈ pcmap do
11 |   |   InsertParent(c);
```

Algorithm 18: InsertParent

Input: y
Output: void

```
1  d(y) := 0;
2  for all x ∈ pred(y) do
3  |   if d(x) ≠ ∞∨p(x) ≠ unknown then
4  |   |   B.enqueue(x);
```

Algorithm 19: ExtractNegativeCycle

Input: pcmap,u,v
Output: neg-cycle

```
1  neg-cycle = ((u,v));
2  x := unknown;
3  y := unknown;
4  for y ≠ v do
5  |   x := pcmap.get(y);
6  |   if x ≠ unknown then
7  |   |   x := p(y);
8  |   |   neg-cycle.add((x,y));
9  |   |   y := x;
10 return reverse(negcycle);
```

Algorithm 20: DeleteSubtree

Input: x,pcmap
Output: pcmap

```
1 if s≠unknown ∧ p(x)≠ unknown then
2 |   B := B\u;
3 |   A := A\u;
4 |   pcmap.put(x → p(x));
5 |   p(x) := unknown;
6 |   for all y ∈ c(x) do
7 |   |   DeleteSubtree(y,pcmap);
8 |   c(p(x)):= c(p(x))\x;
9 return pcmap;
```

Chapter 6

Unifying PDDL and TCA Planning

The two planning problem representations of Timed Concurrent Automata (TCA) and the popular Planning Domain Description Language (PDDL) [42, 21] have important semantic differences, but are still similar enough to be bi-directionally mapped. tBurton uses these mappings in two ways: 1. The mapping from PDDL to TCA enables tBurton to used as a PDDL planner. 2. The mapping from TCA to PDDL enables tBurton to use PDDL planners as sub-planners.

6.1 Mapping PDDL to TCA

Even though tBurton reasons over TCAs, it is still a capable PDDL planner. In order to run tBurton on PDDL problems, we developed a PDDL 2.1 (without numeric fluents) to TCA translator. Here we provide a sketch of this translator.

In order to maintain required concurrency, the translator first uses temporal invariant synthesis [5] to compute a set of invariants. An instance of an invariant identifies a set of ground predicates for which at most one can be true at any given time. We select a subset of these invariant instances that provide a covering of the

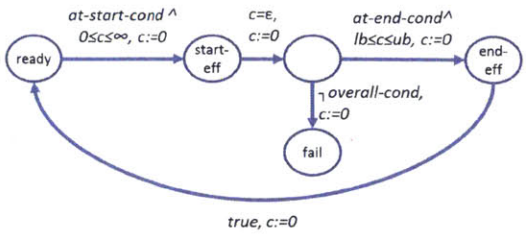


Figure 6-1: An example TCA automaton for a generic PDDL grounded action.

reachable state-space, and encode each invariant instance into an automaton. Each possible grounding of the invariant instance becomes a location in the automata.

Each ground durative action is also translated into an automaton. In Figure 6-1, three of the transitions are guarded by conditions from the corresponding PDDL action, translated into propositional state logic over location variables. Another transition uses ϵ to denote a small-amount of time to pass for the start-effects of the action to take effect, prior to checking for the invariant condition. A fifth transition is used to reset the action.

Finally, the transitions of each invariant-instance based automata is labeled with a disjunction of the states of the ground-action automata that affect its transition.

Inherent in this mapping is an assumption that PDDL durative actions will not self-overlap in a valid plan. Planning with self-overlapping durative actions in PDDL is known to be EXPSPACE, while without such actions is PSPACE [50]. This suggests that tBurton may solve a simpler problem, but the actual complexity of TCA planning with autonomous transitions has yet to be addressed. In the meantime, if the number of duplicate actions can be known ahead of time, they can be added as additional ground-action automata.

6.1.1 Temporal Invariant Synthesis

Intuitively, an *invariant* establishes that the number of relationships a specific object or objects can have with other objects stays fixed. For example, given the relationship defined by the predicate `(at ?package ?location)`, if we show for package `p` that no action can increase the partially grounded predicate, `(at p ?location)`, then we have proven the invariant that a package can only be at one location at a time [18]. Temporal invariant synthesis is similar to the original invariant synthesis work by Helmert, but applies to PDDL 2.1, level 2 durative actions [31].

An invariant has the form $\mathcal{C} = \langle \Phi, \mathrm{F}, \mathrm{V} \rangle$, where Φ is a subset of predicates from the domain, and F and V are disjoint sets whose union is the set of all variables occurring in Φ. An *instance* γ of $\mathcal{C}$, is a function which grounds F. Referring back to the package example, there could be an instance of an automaton for package p_1 and another for p_2.

To make it easier to relate invariants to our model, we further define:

- δ to be a function that grounds V,
- $\Delta = \{\delta_1, ..., \delta_m\}$ to be the set of reachable grounding assignments for V as discovered during the search for invariant $\mathcal{C}$, and
- $\Phi_{\gamma\delta}$ to be a set of ground predicates (possibly singleton) with variables in Φ grounded by γ and δ.

We create an automaton for each instance of an invariant. For a single automaton generated from instance, γ, we create a new location for each set of ground predicates, $\Phi_{\gamma\delta}$, where $\delta \in \Delta$.

Chapter 7

tBurton Benchmarks & Applications

7.1 IPC Benchmarks

Domain	YAHSP3-MT (2014)		POPF (2011)		TFD (2014)		tBurton+TFD	
	#Solved	IPCScore	#Solved	IPCScore	#Solved	IPCScore	#Solved	IPCScore
CREWPLANNING (2011)	**20**	19.88	**20**	**20.00**	**20**	19.85	**20**	**20.00**
DRIVERLOG (2014)	**3**	**1.77**	0	0.00	0	0.00	0	0.00
ELEVATORS (2011)	**20**	11.20	3	2.10	**20**	**18.95**	**20**	**18.95**
FLOORTILE (2011)	**11**	**9.29**	1	0.89	5	5.00	3	3.00
FLOORTILE (2014)	**6**	**5.83**	0	0.00	0	0.00	0	0.00
OPENSTACKS (2011)	**20**	14.47	**20**	16.59	**20**	**19.84**	**20**	**19.84**
PARCPRINTER (2011)	*1*	*1.00*	*0*	*0.00*	*10*	*9.67*	*13*	***11.98***
PARKING (2011)	**20**	15.74	**20**	17.42	**20**	**19.14**	**20**	**19.14**
PARKING (2014)	**20**	**17.96**	12	9.16	**20**	16.18	**20**	16.18
PEGSOL (2011)	**20**	18.52	19	**18.77**	19	18.42	18	17.38
SATELLITE (2014)	***20***	***17.46***	*4*	*3.67*	*17*	*12.57*	*19*	*16.26*
SOKOBAN (2011)	**10**	**8.69**	3	2.54	5	4.94	3.00	2.54
STORAGE (2011)	**7**	**6.54**	0	0.00	0	0.00	0	0.00
STORAGE (2014)	**9**	**8.41**	0	0.00	0	0.00	0	0.00
TMS (2011)	*0*	*0.00*	*5*	***5.00***	*0*	*0.00*	***6***	*3.77*
TMS (2014)	*0*	*0.00*	*0*	*0.00*	*0*	*0.00*	*1*	***1.00***
TURNANDOPEN (2011)	*0*	*0.00*	*9*	*8.47*	*19*	*16.53*	***20***	***17.03***
TURNANDOPEN (2014)	0	0.00	0	0.00	**6**	**6.00**	**6**	**6.00**
TOTALS:	187	156.74	116	104.61	181	167.09	**189**	**173.07**

Table 7.1: Benchmark results on IPC domains

We benchmarked tBurton on a combination of IPC 2011 and IPC 2014 domains. Temporal Fast Downward (TFD) from IPC 2014 was used as an 'off-the-shelf' sub-

planner for tBurton because it was straight-forward to translate $TCAs$ to the SAS representation used by TFD [30]. For comparison, we also benchmarked against YAHSP3-MT from IPC 2014, POPF2 from IPC 2011, and TFD from IPC 2014, the winner or runners up in the 2011 and 2014 temporal satisficing track (Table 7.1). Each row represents a domain. Columns are grouped by planner and show the number of problems solved (out of 20 for each domain) and the IPC score (out of 20, based on minimizing make-span). Rows with interesting results between TFD and tBurton are italicized, and the best scores in each domain are in bold. The tests were run with scripts from IPC 2011 and were limited to 6GB memory and 30 minute runtime.

In general, tBurton is capable of solving approximately the same number of problems as TFD with the same quality, but for problems which admit some acyclic causal factoring (parcprinter, satellite, and TMS), tBurton performs particularly well. On domains which have no factoring, tBurton's high-level search provides no benefit and thus degrades to using the sub-planner. This often resulted in tBurton receiving the same score as its subplanner, TFD (elevators, parking, openstack). While the same score often occurs when TFD is already sufficiently fast enough to solve the problem, there are a few domains where tBurton's processing overhead imposes a penalty (floortile, sokoban).

A feature of using tBurton is that it is capable of solving problems with required concurrency. For TMS, tBurton is able to consistently solve more problems than the other planners. However, the IPCScore of tBurton is lower than POPF, perhaps because the goal-ordering strategy used by tBurton does not inherently minimize the make-span like the temporal relaxed planning graph used by POPF.

While benchmarking on existing PDDL domains is a useful comparison, it is worth noting that these problems do not fully demonstrate tBurton's capabilities. In particular, none of these domains have temporally extended goals, and all actions have a single duration value instead of an interval. We look forward to more comprehensive

testing with both existing PDDL domains and developing our own benchmarks.

7.2 Demonstrations

7.3 Cassini Main Engine

In order to demonstrate tBurton's applicability to real-world scenarios, such as the control of spacecraft, we applied tBurton to the problem of firing the Cassini main engines given a stuck valve. Presented in figure 1-3, the Cassini example requires 387 automata to model all of its valves and flows, with each automaton requiring an average of 4 states. When factored into a causal graph, the problem contained 210 factors. Represented in tBurton's native TCA encoding, the problem took tBurton (with TFD subplanner) approximately 2.6 seconds to solve. It took TFD alone 18 minutes and POPF over a day.

7.3.1 VEX Castle Domain

In order to demonstrate tBurton's scalability as well as versatility in assembly planning, we ran tBurton on a synthetic domain representing the assembly of a VEX toy constructor set into the shape of a 2D castle (Figure 7-1). The problem consisted of 71 equal-length metal bars that needed to pinned at the ends. To simulate a manufacturing scenario, some subassemblies and their assembly order were dictated ahead of time (Figure 7-2). The number of workers available was left as an adjustable parameter, and tested for 1, 2, 4, and 16 workers. tBurton was capable of solving all of these problems in approximately 30 minutes or less, while the leading temporal planner, POPF, was not capable of completing the assembly plan for even 1 worker in 30 minutes.

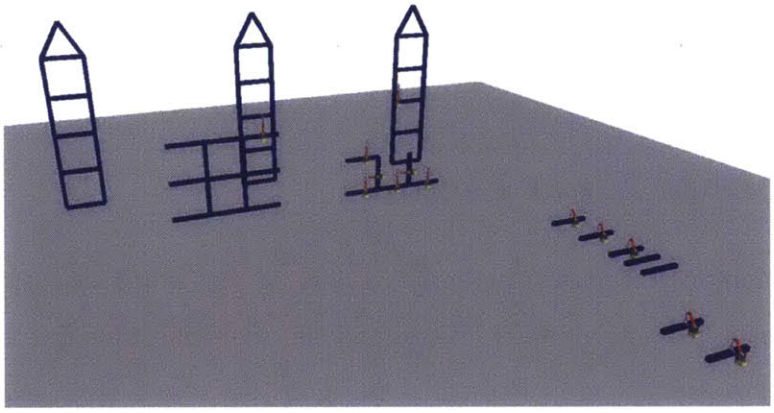


Figure 7-1: An Open-Rave 3D Simulation of VEX construction bars being assembled into a castle.

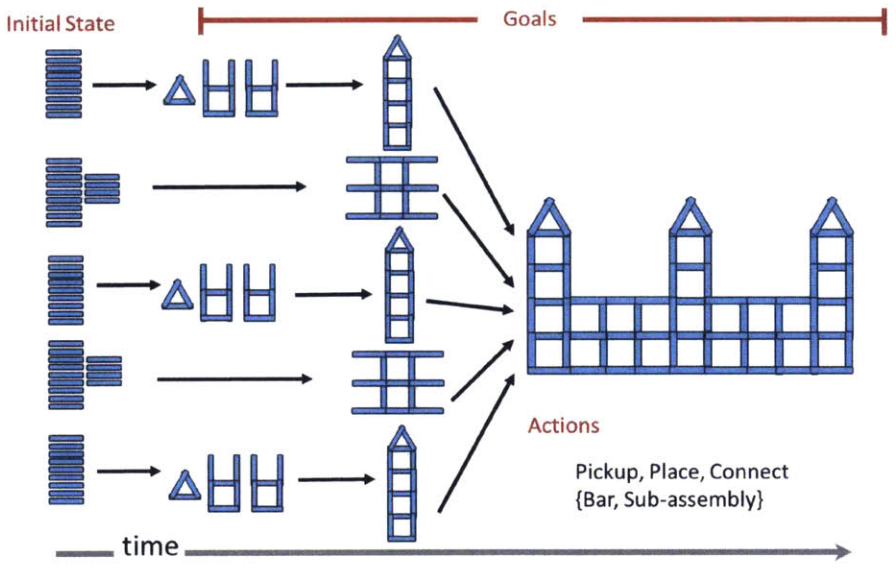


Figure 7-2: A pictoral representation of the State Plan used to encode the VEX Castle Assembly problem.

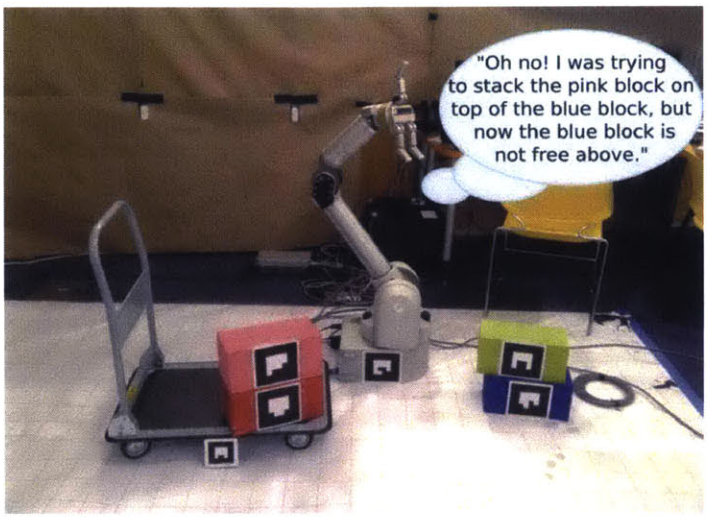


Figure 7-3: In a real-time physical demonstration of block stacking, tBurton provided plans for execution-time recovery.

7.3.2 Block Stacking and Recovery

We also deployed tBurton in a real-time physical demonstration of block stacking. For this demonstration we used a Barrett Whole Arm Manipulator (WAM) to move color-coded brick-sized blocks (Figure 7-3). The objective was to place the pink block ontop of the blue block. Object identification and tracking was done through ARTags and an HD camera system. Replanning was triggered by Pike, an execution monitoring capability that took the ground-facts from object tracking and determined whether an execution failure was pending [39]. Once a potential failure was detected, tBurton was called to sequence a new plan for the WAM. [1]

[1]Thank you to the hardwork of Steve Levine, Pedro Santana, and Erez Karpas, for the design and programming of this scenario.

Chapter 8

Future Work & Conclusions

8.1 Extending TCA Planning to Uncertainty and Risk

Until now, we've assumed that we can control how long a TCA dwells in any state and that the outcome of a transition is deterministic. In the real world, this is not always the case.

An automatic timed-transition should be able to occur at any time allowed by the clock-constraint in its guard. In our projector example, when the projector is first turned-on, its transition from `WarmUp` to `On` could be guarded by a clock constraint $0 \leq c \leq 30$. We currently let the planner decide when, in this interval, the projector turns on, but in the real-world, this transition is actually dictated by thermodynamic constraints. In the real world, this decision is 'made by' the projector.

Transitions are also not necessarily deterministic. In the projector example, one could imagine the remote control from which I am issuing commands does not always transmit, and therefore the projector automaton does not transition as expected. In a more relatable example, on one's drive home from work, they may reach home safely

99% of the time, but 1% of the time get in a car accident. The outcome of 'driving home' is uncertain.

In this chapter we discuss the foundations for future work that extend tBurton to resolve a form of temporal uncertainty. We base this work on previous work in *risk-aware* reasoning [45], which introduce *chance-constraints* as upper-bounds on the probability of failure. By exerting only enough effort to satisfy these chance-constraints, we can potentially save on computation time in comparison to alternatives such as minimizing the probability of failure.

8.2 Temporal Uncertainty through Rubato

In our state plan representation, the passage of time is always associated with an episode. The state-constraint associated with a particular episode must be maintained for the duration of that episode, and the allowable values for that duration are lower and upper-bounded by a temporal constraint.

In order to represent temporal uncertainty, we introduce an alternative bound on that duration based on a probability distribution. The distribution represents the probability that the system we are planning for will choose a particular duration for that episode. For example, when driving from A to B, traffic might dictate that we will be driving for an average of 5 minutes with a standard-deviation of 1 minute. The distribution of durations, in general, can take any form.

The objective of the planner is then to choose all of the other durations under its control, such that the plan will succeed within the probability of failure imposed by a set of chance-constraints.

Formally, the temporal-network underlying this type of plan is refered to as a Simple Temporal Network with Uncertainty (STNU). For a STN, we were interested in the question of temporal consistency. For a STNU, we are interested in whether it

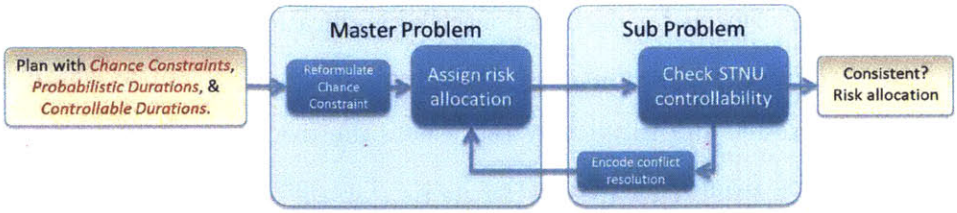


Figure 8-1: The master and sub problem architecture of the Rubato strong controllability checking algorithm.

is *controllable*. There are several different forms of controllability an STNU can take: weak, strong, and dynamic. The variation depends on when we find out what the durations have been selected for the uncontrollable durations. In weak controllability, we assume an oracle tells us how long each uncontrollable duration takes, and ask whether the remaining controllable durations can be selected to make the STNU temporally consistent. In strong controllability, we want to know regardless of how long the uncontrollable durations take, we can select the controllable durations to make the STNU temporally consistent. Finally, in dynamic controllability, we want to know during a hypothetical execution, if we are told how long each uncontrollable duration takes as they complete, could we select the controllable durations on-demand so the STNU is temporally consistent.

For this discussion of incorporating temporal uncertainty into tBurton, we are interested in checking whether our plans are *strongly controllable*. The major motivation for this choice is the readily available Rubato algorithm [56], which was designed to efficiently answer the question of strong controllability.

8.2.1 Rubato

We take a moment to elaborate on Rubato to elucidate the future work.

Rubato determines whether an STNU is strongly controllable by iteratively solving a master and sub-problem. The master problem uses a non-linear solver to choose

an upper-bound duration for each uncontrollable duration, effectively cuting off the tails of the probability distributions. The probability mass removed must satisfy the chance constraint, and intuitively represents the risk of disallowing these uncontrollable durations from ever taking too long. The subproblem then interprets the values chosen for those upper-bound durations as interval-based uncontrollable durations, allowing known STNU solvers to determine whether the problem is strongly controllable.

Figure 8-2 shows a simple progression of a chance-constrained STNU through Rubato's master and sub problem architecture. The STNU consists of 4 uncontrollable durations and a chance-constraint constraint, which requires the entire plan to take no more than 60 minutes, with a 20% probability of failure.

Rubato's unique approach to solving a chance-constrained STNU (CC-STNU) makes it one of the fastest controllability checkers available. However, it can still take 10 seconds to check for the consistency of a problem with 100 uncontrollable durations, and over 2 hours to check for the consistency of a problem with 300 uncontrollable durations [56].

8.2.2 Rubato & tBurton

For tBurton to support temporal uncertainty, ITCv2 needs to be replaced with Rubato (Figure 8-3). The problem with this replacement is that Rubato is currently significantly slower than ITCv2 due the probabilistic reasoning it must perform. In order to make tBurton fast enough to be useful, Rubato needs to be sped up.

We see straight forward and complementary ways to improve Rubato. One approach is to add a pre-processor to Rubato that intercepts and handles changes to controllable constraints. This pre-processor would only invoke Rubato when a controllable constraint intrudes on durations allocated to uncontrollable durations. The

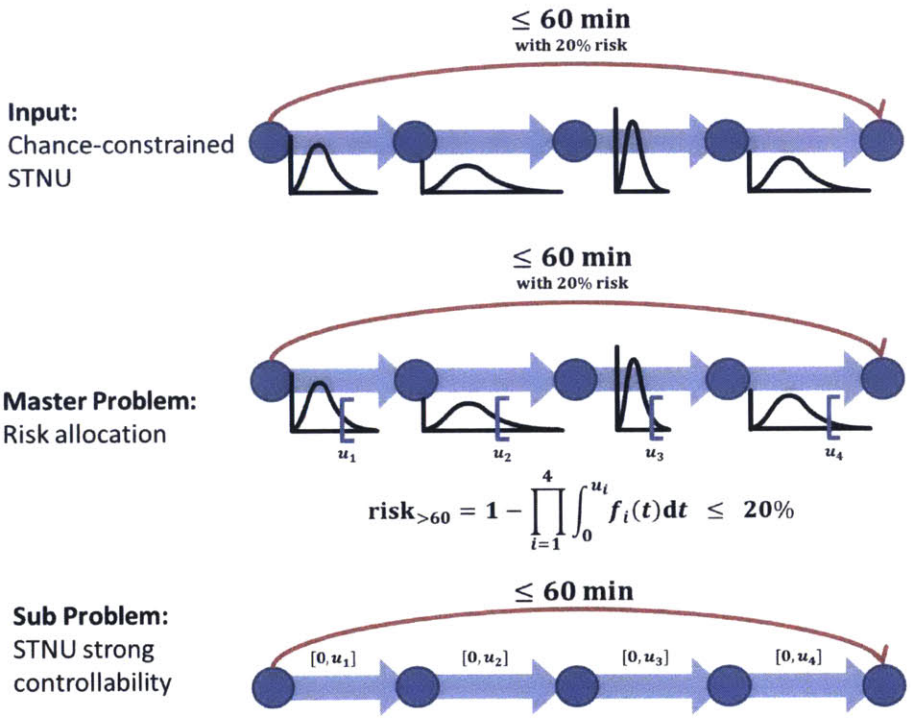


Figure 8-2: Given a chance-constrained STNU, Rubato iteratively solves the problem by allocating risk in a master problem, and then solving the simplified STNU as a strong controllability sub problem.

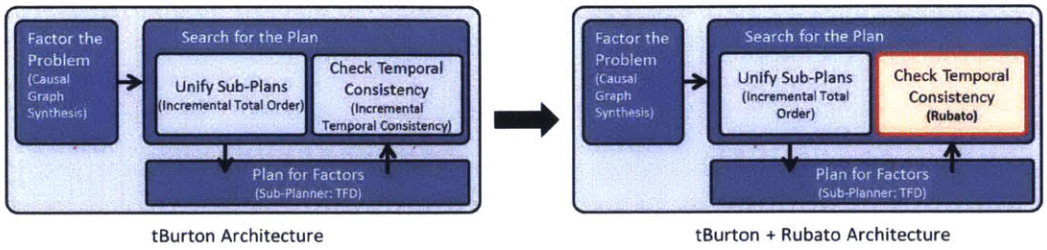


Figure 8-3: A comparison of the original tBurton architecture and the new architecture in which Rubato replaces ITCv2.

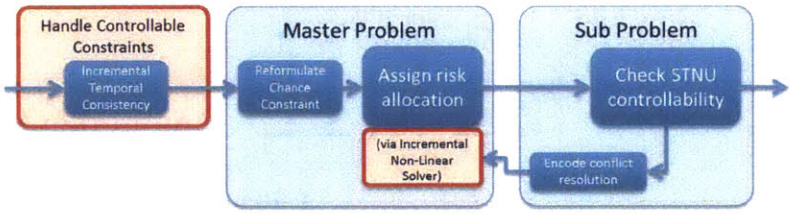


Figure 8-4: In order to make Rubato faster, we propose handling controllable constraints separately and switching to an incremental NLP solver.

other approach is to make Rubato incremental by replacing its non-linear solver with an incremental non-linear solver (Figure 8-4).

8.3 Conclusion

This thesis presented tBurton, a planner that uses a novel combination of causal-graph factoring, timeline-based regression, and heuristic forward search to plan for networked devices. It is capable of supporting a set of problem features not found before in one planner, and is capable of doing so competitively. Furthermore, tBurton can easily benefit from advancements in state-of-the art planning by replacing its sub-planner. The planning problem tBurton solves assumes devices with controllable durations, but we often have little control over the duration of those transitions. In future work we plan to add support for uncontrollable durations.

Bibliography

[1] James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[2] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[3] Eyal Amir and Barbara Engelhardt. Factored planning. In *IJCAI*, volume 3, pages 929–935. Citeseer, 2003.

[4] J Benton, Amanda Jane Coles, and Andrew Coles. Temporal planning with preferences and time-dependent continuous costs. In *ICAPS*, volume 77, page 78, 2012.

[5] Sara Bernardini and David E Smith. Automatic synthesis of temporal invariants. In *SARA*, 2011.

[6] A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.

[7] Mark S. Boddy. Imperfect match: Pddl 2.1 and real applications. *J. Artif. Intell. Res. (JAIR)*, 20:133–137, 2003.

[8] Ronen I Brafman and Carmel Domshlak. Structure and complexity in planning with unary operators. *J. Artif. Intell. Res.(JAIR)*, 18:315–349, 2003.

[9] Ronen I Brafman and Carmel Domshlak. Factored planning: How, when, and when not. In *AAAI*, volume 6, pages 809–814, 2006.

[10] B.V. Cherkassky and A.V. Goldberg. Negative-cycle detection algorithms. *Mathematical Programming*, 85(2):277–311, 1999.

[11] Steve Chien, G Rabideau, R Knight, Robert Sherwood, Barbara Engelhardt, Darren Mutz, T Estlin, Benjamin Smith, F Fisher, T Barrett, et al. ASPEN–automated planning and scheduling for space mission operations. In *Space Ops*, pages 1–10, 2000.

[12] Seung Hwa Chung and Brian C Williams. *A decomposed symbolic approach to reactive planning.* PhD thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, 2003.

[13] Amanda Jane Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. In *ICAPS*, pages 42–49, 2010.

[14] William Cushing, Subbarao Kambhampati, Daniel S Weld, et al. When is temporal planning really temporal? In *Proceedings of the 20th international joint conference on Artifical intelligence*, pages 1852–1859. Morgan Kaufmann Publishers Inc., 2007.

[15] Lisa Dale. *The true cost of wildfire in the Western US.* Western Forestry Leadership Coalition, 2009.

[16] R. Dechter. *Constraint processing.* Morgan Kaufmann, 2003.

[17] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial intelligence*, 49(1-3):61–95, 1991.

[18] S. Edelkamp and M. Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. *Recent Advances in AI Planning*, pages 135–147, 2000.

[19] R.E. Fikes and N.J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1972.

[20] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[21] M. Fox and D. Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003.

[22] Jeremy Frank and Ari Jónsson. Constraint-based attribute and interval planning. *Constraints*, 8(4):339–364, 2003.

[23] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):251–281, 2000.

[24] Michael Gelfond and Vladimir Lifschitz. Action languages. *Electronic Transactions on AI*, 3(16), 1998.

[25] A. Gerevini and D. Long. Plan constraints and preferences in pddl3. *Technichal Report, Department of Electronics for Automation, University of Brescia, Italy*, 2005.

[26] M. Ghallab, A. Howe, D. Christianson, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl—the planning domain definition language. *AIPS98 planning committee*, 78(4):1–27, 1998.

[27] Andrew V Goldberg. Scaling algorithms for the shortest paths problem. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 222–231. Society for Industrial and Applied Mathematics, 1993.

[28] Andrew V Goldberg and Tomasz Radzik. A heuristic improvement of the bellman-ford algorithm. *Applied Mathematics Letters*, 6(3):3–6, 1993.

[29] M. Helmert. A planning heuristic based on causal graph analysis. ICAPS, 2004.

[30] M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26(1):191–246, 2006.

[31] M. Helmert. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.

[32] J. Hoffmann. Ff: The fast-forward planning system. *AI magazine*, 22(3):57, 2001.

[33] Michel Donald Ingham. *Timed model-based programming: Executable specifications for robust mission-critical sequences.* PhD thesis, Massachusetts Institute of Technology, 2003.

[34] Rune Mller Jensen. *Efficient BDD-based planning for non-deterministic, fault-tolerant, and adversarial domains.* PhD thesis, Citeseer, 2003.

[35] H. Kautz and B. Selman. Unifying sat-based and graph-based planning. In *INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 16, pages 318–325. LAWRENCE ERLBAUM ASSOCIATES LTD, 1999.

[36] Elena Kelareva, Olivier Buffet, Jinbo Huang, and Sylvie Thiébaux. Factored planning using decomposition trees. In *IJCAI*, pages 1942–1947, 2007.

[37] Valerie King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 81–89. IEEE, 1999.

[38] Stavros G Kolliopoulos and Clifford Stein. Finding real-valued single-source shortest paths in¡ i¿ o¡/i¿(¡ i¿ n¡/i¿¡ sup¿ 3¡/sup¿) expected time. *Journal of Algorithms*, 28(1):125–141, 1998.

[39] Steven James Levine. *Monitoring the execution of temporal plans for robotic systems.* PhD thesis, Massachusetts Institute of Technology, 2012.

[40] Stefan Lewandowski. Shortest paths and negative cycle detection in graphs with negative weights. i, the bellman-ford-moore algorithm revisited. 2010.

[41] Dennis L Matson, Linda J Spilker, and Jean-Pierre Lebreton. The cassini/huygens mission to the saturnian system. In *The Cassini-Huygens Mission*, pages 1–58. Springer, 2003.

[42] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl-the planning domain definition language. 1998.

[43] Nicola Muscettola, Paul Morris, and Ioannis Tsamardinos. Reformulating temporal plans for efficient execution. In *In Principles of Knowledge Representation and Reasoning.* Citeseer, 1998.

[44] Akimitsu Ono and Shin-ichi Nakano. Constant time generation of linear extensions. In *Fundamentals of Computation Theory*, pages 445–453. Springer, 2005.

[45] Masahiro Ono, Brian C. Williams, and Lars Blackmore. Probabilistic planning for continuous dynamic systems under bounded risk. *Journal of Artificial Intelligence Research*, 46:511–577, 2013.

[46] James Paterson, Eric Timmons, and Brian C. Williams. A scheduler for actions with iterated durations. In *AAAI-14*, 2014.

[47] J.S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for adl. In *proceedings of the third international conference on knowledge representation and reasoning*, pages 103–114. Citeseer, 1992.

[48] G Ramalingam, Junehwa Song, Leo Joskowicz, and Raymond E Miller. Solving systems of difference constraints incrementally. *Algorithmica*, 23(3):261–275, 1999.

[49] Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks revisited. In *AAAI*, volume 8, pages 975–982, 2008.

[50] Jussi Rintanen. Complexity of concurrent temporal planning. In *ICAPS*, pages 280–287, 2007.

[51] Gabriele Röger, Patrick Eyerich, and Robert Mattmüller. TFD: A numeric temporal extension to fast downward. ipc 2008 short pape rs, 2008.

[52] I. Shu, R. Effinger, and B. Williams. Enabling fast flexible planning through incremental temporal reasoning with conflict extraction. *Proce. ICAPS-05, Monterey*, 2005.

[53] R.E. Tarjan. Shortest paths. Technical report, AT&T Bell Laboratories, Murry Hill, NJ, 1981.

[54] Mikkel Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 112–119, New York, NY, USA, 2005. ACM.

[55] Vincent Vidal. YAHSP2: Keep it simple, stupid. *The 2011 International Planning Competition*, page 83, 2011.

[56] Brian Wang, Andrew. Williams. Chance-constrained scheduling via conflict-directed risk allocation. 2015.

[57] B.C. Williams and P. Nayak. A reactive planner for a model-based executive. In *International Joint Conference on Artificial Intelligence*, volume 15, pages 1178–1185. LAWRENCE ERLBAUM ASSOCIATES LTD, 1997.

[58] B.C. Williams and P.P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 971–978, 1996.

[59] Brian C. Williams. Doing time: Putting qualitative reasoning on firmer ground. In *Proceedings of the National Conference on Artificial Intelligence*, pages 105–113, Philadelphia, PA, 1986.

[60] Brian C. Williams and Robert Ragno. Conflict-directed a* and its role in model-based embedded systems. *Special Issue on Theory and Applications of Satisfiability Testing, Journal of Discrete Applied Math*, 155(12):1562–1595, jun 2003.

[61] Håkan LS Younes and Reid G Simmons. Vhpop: Versatile heuristic partial order planner. *J. Artif. Intell. Res.(JAIR)*, 20:405–430, 2003.